M. Nardone, C. Scarioni, *Pro Spring Security*
https://doi-org.ezproxy.sfpl.org/10.1007/979-8-8688-0035-1_5

5. Web Security

Massimo Nardone[1] and Carlo Scarioni[2]
(1) HELSINKI, Finland
(2) Surbiton, UK

This chapter shows how to build a Java web application using Spring Security 6 in Spring Boot 3. You see the inner workings of the security filter chain and the different metadata options at your disposal to define security constraints in your application.

Let's build your Java web application using Spring Security 6 in Spring Boot 3, and please make sure you're using Java 17+, as the baseline for Spring Boot 3 and Spring Security 6 is now Java 17. Java 20 is used in this demo.

The following steps build the simple Spring Security Maven web application project.

1. Create a new Spring Security Spring Boot 3 project, including Spring Security and Spring Web dependencies, using the start.spring.io Spring Initializr website.
2. Configure the users and roles that will be part of the system.
3. Configure the URLs that you want to secure.
4. Create all needed Java and web files.
5. Run the Spring Security 6 project using the external Tomcat Server 10.

First, you create a new Spring project named pss01_Security using the Spring Initializr web tool at `https://start.spring.io/`, as shown in Figure 5-1.

Java 20, Maven, and JAR, with Spring Security and Web, are dependencies in this example.

Once the project is generated, unzip the file and open the project with your IDE tool.

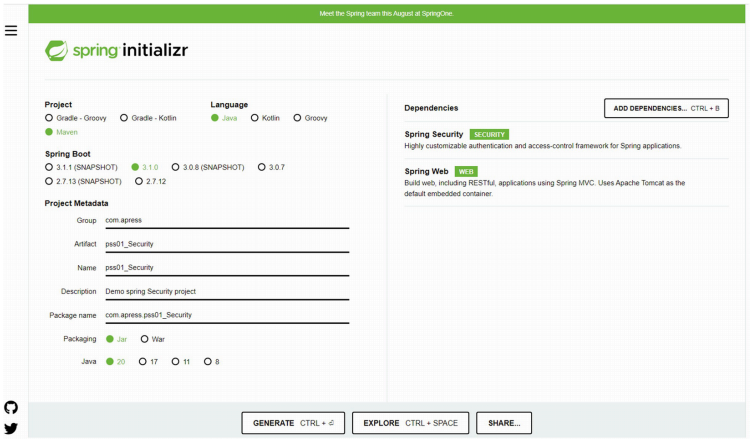

Figure 5-1 New Spring project using Spring Initializr

The new project files and the pss01_Security project structure are shown in Figure 5-2.

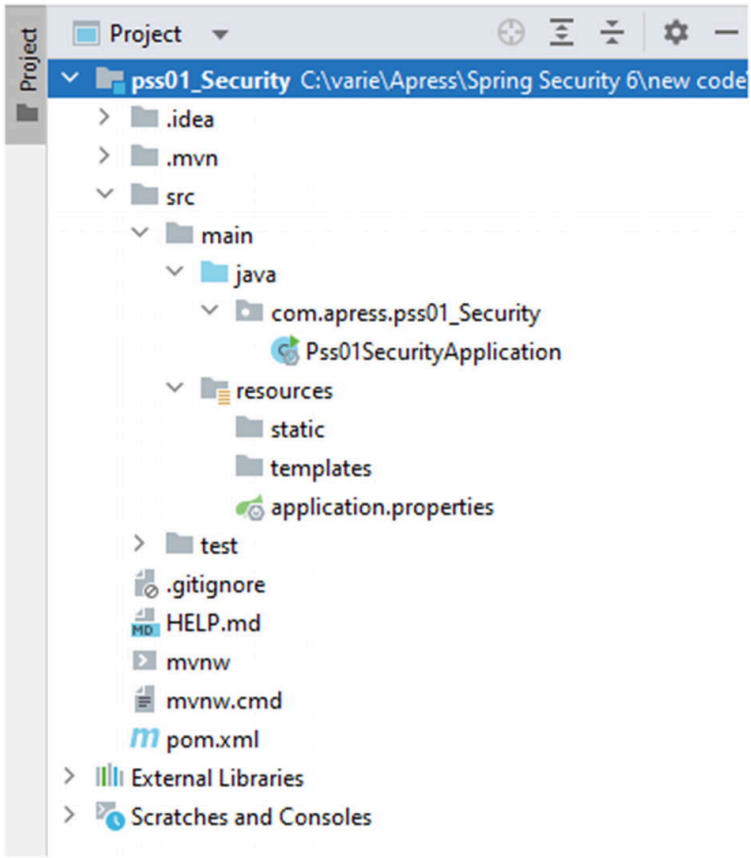

Figure 5-2 New Spring project structure

> **Note** Implementing the Spring Security in a Spring application using XML- or Java-based configurations is possible. This chapter uses Java configuration for your Spring Security web application since it is hardly suggested to use XML configuration as minimum as possible.

If Spring Security is in the classpath, Spring Boot automatically secures all HTTP endpoints with Basic authentication, generating a security password to be used as a credential with "user" as the username, as shown in Figure 5-3.

Figure 5-3 Running the new Spring project

This means that if you type **localhost:8080**, Spring requires that you enter **user** as the username and **e6fd5a38-b7a8-4d55-b47a-9ece6e3341fa** as the password to log in, as shown in Figure 5-4.

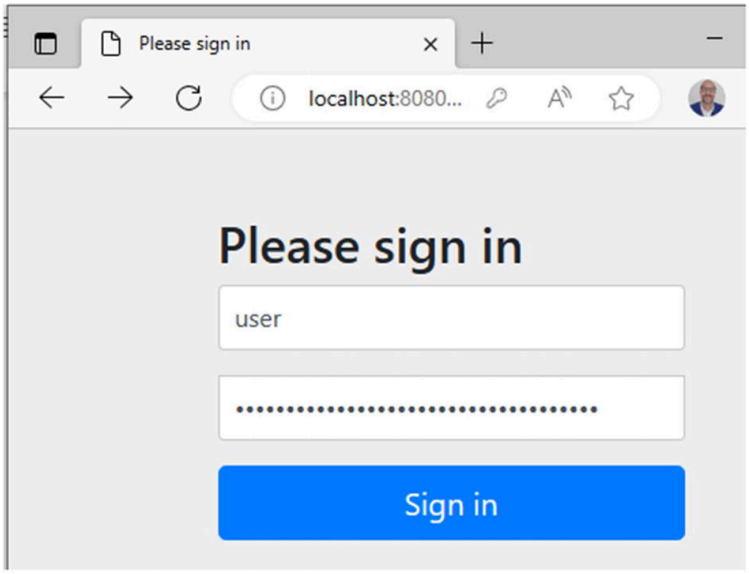

Figure 5-4 Secure Spring application with login page

Since this web application is based on Spring MVC, you need to configure Spring MVC and set up view controllers to expose the HTML templates that you will create later.

Let's create a simple controller to get a simple "Welcome to Spring Security 6" message when entering the right login information, as shown in Listing 5-1.

```
package com.apress.pss01_Security;


import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;
```

```
@RestController




public class UserController {

    @GetMapping ("/welcome")




    public String welcome() {

        return "Welcome to Spring Security 6";

    }




}
```

Listing 5-1
A Simple UserController Java Class

If you enter the right username and password, you get the "Welcome to Spring Security 6" message, as shown in Figure 5-5.

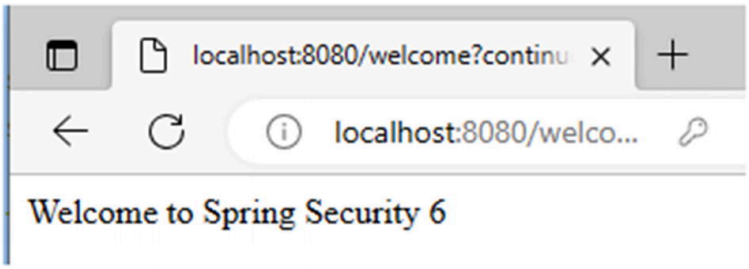

Figure 5-5 Successful login message

Let's add some more logic to our code.

First, let's look at the new `pom.xml` file generated when creating the new Spring Boot 3 and Spring Security 6 projects, as shown in Listing 5-2.

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns:="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instanc

    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"

    <modelVersion>4.0.0</modelVersion>

    <parent>

       <groupId>org.springframework.boot</groupId>

       <artifactId>spring-boot-starter-parent</artifactId>

       <version>3.1.0</version>

       <relativePath/> <!-- lookup parent from repository -->

    </parent>

    <groupId>com.apress</groupId>

    <artifactId>pss01_security</artifactId>

    <version>0.0.1-SNAPSHOT</version>

    <name>pss01_security</name>

    <description>Spring Security demo</description>

    <properties>

       <java.version>20</java.version>

    </properties>

    <dependencies>
```

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



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

```
        <groupId>org.thymeleaf.extras</groupId>

        <artifactId>thymeleaf-extras-springsecurity6</artifactId>

        <version>3.1.1.RELEASE</version>

        </dependency>

     </dependencies>




</project>
```

Listing 5-2
pom.xml

We used Thymeleaf, a Java template engine for processing and creating HTML, XML, CSS, JavaScript, and plain text.

Configuring the new Spring Security 6 Project

To activate Spring Security web project configuration in your Maven web application, you need to configure a particular servlet filter that takes care of preprocessing and postprocessing the requests and managing the required security constraints.

The next example defines two users, but only the “Admin” role is authorized to access the secured resource called authenticated.html.

Let’s start building the Spring Security Maven web application.

First, make sure that all the tools and directories are created as described previously.

Next, create simple HTML files under a new project directory called `src/main/resources/templates/`.

Your project utilizes two HTML pages.

- `welcome.html`, which is the starting welcome web page of the project

- `authenticated.html`, which is the admin web page to access when the user successfully logs in

The `welcome.html` page is shown in Listing 5-3.

```
<!DOCTYPE html>

<html xmlns:="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org">

<html lang="en">

<head>

    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

    <title>Spring Security 6 authentication example!</title>

</head>

<body>



<div th:if="${param.error}">

    Invalid username and password.

</div>

<div th:if="${param.logout}">

    You have been logged out.

</div>



<h2>Welcome to Spring Security 6 authentication example!</h2>
```

```
<p>Click <a th:href="@{/authenticated}">here</a> to get authenticated!</p>




</body>

</html>
```

Listing 5-3

welcome.html

The `welcome.html` page only displays a welcoming message and provide the link to the authenticated page, `/authenticated`.

Let's now create the `authenticated.html` page; see Listing 5-4.

```
<!DOCTYPE html>

<html xmlns:="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org"

      xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-springsecurity6">

<head>

    <title>Spring Security 6 authentication example</title>

</head>

<body>

<h2>Welcome to Spring Security 6 authentication example!</h2>
```

```
<h2 th:inline="text">You are an authenticated user: <span th:remove="tag" sec:authentication="name">thy

<p>click <a th:href="@{/logout}">here</a> to logout!!</p>

</body>

</html>
```

Listing 5-4
authenticated.html

Next, you need to define the Java classes needed for your example.

- Under package controller: `UserController`
- Under package configuration: `SecurityConfiguration`

Let's create the two Java packages where your Java classes are located.

- `package com.apress.pss01_security.configuration`
- `package com.apress.pss01_security.controller;`

Create the `UserController` Java class under the `com.apress.pss01_security.controller` package, as shown in Listing 5-5.

```
package com.apress.pss01_security.controller;

import org.springframework.stereotype.Controller;

import org.springframework.ui.ModelMap;

import org.springframework.web.bind.annotation.GetMapping;
```

```
@Controller

public class UserController {



    @GetMapping("/")

    public String homePage() {

        return "welcome";

    }



    @GetMapping("/welcome")

    public String welcomePage() {

        return "welcome";

    }



    @GetMapping ("/authenticated")

    public String AuthenticatedPage() {

        return "authenticated";

    }
```

```
    @GetMapping ("/logout")
    public String logoutPage() {
    return "redirect:/welcome";
    }

}
```

Listing 5-5
UserController Java Class

Note that, for web security, it doesn't matter if you use a Spring MVC controller as you do here, if you use simple servlets as you did in Chapter 3, or if you use any other servlet-based framework for developing your application. Remember that, at the core, the web part of Spring Security attaches itself to the standard Java servlet filter architecture. So, if your application uses servlets and filters, you can leverage Spring Security's web support.

Since Spring Framework 4.3, there are new HTTP mapping annotations based on `@RequestMapping`.

- `@GetMapping`
- `@PostMapping`
- `@PutMapping`
- `@DeleteMapping`
- `@PatchMapping`

For instance, `@GetMapping` is a specialized version of the `@RequestMapping` annotation, which acts as a shortcut for `@RequestMapping(method = RequestMethod.GET)`. `@GetMapping` annotated methods to handle the HTTP GET requests matched with a certain given URI expression.

As all developers know, MVC applications aren't service-oriented, which means a view resolver will render the final views based on data received from the controller.

RESTful applications are designed to be service-oriented and return raw data, generally JSON/XML. Since these applications don't do any view rendering, there are no view resolvers. The controller is typically expected to send data directly via the HTTP response.

The UserController Java class, via Spring MVC, does the following.

- Intercepts any incoming request
- Converts the payload of the request to the internal structure of the data
- Sends the data to model for any needed further processing
- Gets processed data from the model, and advances it to the view for rendering

In our example, the UserController Java class returns a "welcome" view. The view resolver tries to resolve the welcome.html page in the templates folder.

Let's analyze our next Java class, `SecurityConfiguration`.

Chapter 2 explained how to enable Spring Security 6 using the annotation named @EnableWebSecurity without using the WebSecurityConfigurerAdapter class, and introduced in that chapter the Java Spring Security configuration class named `SecurityConfiguration`, which utilizes the @EnableWebSecurity annotation to help configure Spring Security–related beans, such as WebSecurityConfigurer or SecurityFilterChain.

The following describes what `SecurityConfiguration` does in this example.

- It creates two demo in-memory users via `UserDetailsService` named "user" and "admin," which are authorized to access a secure resource of the project so that only the admin can access the secured "Authenticated" web resource.
- It uses BCryptPasswordEncoder to encode the user passwords for added security.
- It configures the SecurityFilterChain bean with the username/password Basic authentication mechanism to authenticate the users.

Listing 5-6 shows the SecurityConfiguration Java class.

```
package com.apress.pss01_security.configuration;



import org.springframework.context.annotation.Bean;
```

```
import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.Customizer;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.core.userdetails.User;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.security.provisioning.InMemoryUserDetailsManager;

import org.springframework.security.web.SecurityFilterChain;



@Configuration

@EnableWebSecurity



public class SecurityConfiguration {



    @Bean

    public SecurityFilterChain filterChain1(HttpSecurity http) throws Exception {
```

```
        http

                .authorizeHttpRequests((authorize) -> authorize

                        .requestMatchers("/", "/welcome").permitAll()

                        .requestMatchers("/authenticated").hasRole("ADMIN")

                        .anyRequest().denyAll()




                )

                .csrf(Customizer.withDefaults())

                .formLogin(withDefaults())




                .logout((logout) -> logout

                        .logoutSuccessUrl("/welcome")

                        .deleteCookies("JSESSIONID")

                        .invalidateHttpSession(true)

                        .permitAll()

                );




           return http.build();

    }
```

```
@Bean
public UserDetailsService userDetailsService(){

    UserDetails user = User.builder()
            .username("user")
            .password(passwordEncoder().encode("userpassw"))
            .roles("USER")
            .build();

    UserDetails admin = User.builder()
            .username("admin")
            .password(passwordEncoder().encode("adminpassw"))
            .roles("ADMIN")
            .build();

    return new InMemoryUserDetailsManager(user, admin);
}
```

```
    @Bean
    public static PasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }
}
```

Listing 5-6
SecurityConfiguration.java

Spring Security allows you to model your authorization at the request level. In the example, the /welcome page is permitted to all pages under /admin, requiring one authority, while all other pages require authentication.

By default, Spring Security requires that every request be authenticated. That said, whenever you use an HttpSecurity instance, you must declare your authorization rules.

Whenever you have an HttpSecurity instance, you should at least do the following.

```
http
    .authorizeHttpRequests((authorize) -> authorize
        .anyRequest().authenticated()
    )
```

In our example, the following applies.

- "/" and "/welcome" are permitted to all.
- "/authenticated" can only be accessed when presenting a user with the "Admin" role via the .hasRole("ADMIN") declaration.
- .logout((logout) -> logout is permitted to all and, if utilized, requests the welcome.html page.

More information is at **https://docs.spring.io/spring-security/reference/servlet/authorization/authorize-http-requests.html**.

In an application where end users can log in, it is important to consider how to protect against cross-site request forgery (CSRF). Spring Security protects against CSRF attacks by default for unsafe HTTP methods, such as a POST request, so no additional code is necessary.

In a CSRF attack, a hacker can modify the state of any HTTP method (GET or POST), redirecting the client, for instance, by clicking a modified link to a non-secure web page with the result of stealing a user's sensitive information.

Let's look at CSRF and how to prevent CSRF attacks using Spring Security. The following are common CSRF attacks.

- An HTTP GET request convinces the victim to click a fake GET link to get sensitive information (username/password, etc.)
- An HTTP POST request is the same as GET but uses the POST method.

To use the Spring Security CSRF protection, you must ensure the right HTTP methods (PATCH, POST, PUT, DELETE, etc.) can modify the state.

As of Spring Security 6.0.1 and Spring Boot 3.0.2, the method CookieCsrfTokenRepository.saveToken only gets called when the CsrfFilter calls deferredCsrfToken.get(), which only gets called on POST, PUT, PATCH, and DELETE methods. Unfortunately, the client must expect a failure on the first request under the current implementation. Under previous versions of Spring Security, you could count on the token's cookie being included in the response to GET, HEAD, or OPTIONS requests.

For more information, refer to

https://docs.spring.io/spring-security/reference/6.1-SNAPSHOT/servlet/exploits/csrf.html

In the example, the default configuration is specified explicitly using `.csrf(Customizer.withDefaults())`. The default form login is used by adding line `.formLogin(withDefaults());`.

The structure of your new Spring Security 6 project should look like Figure 5-6.

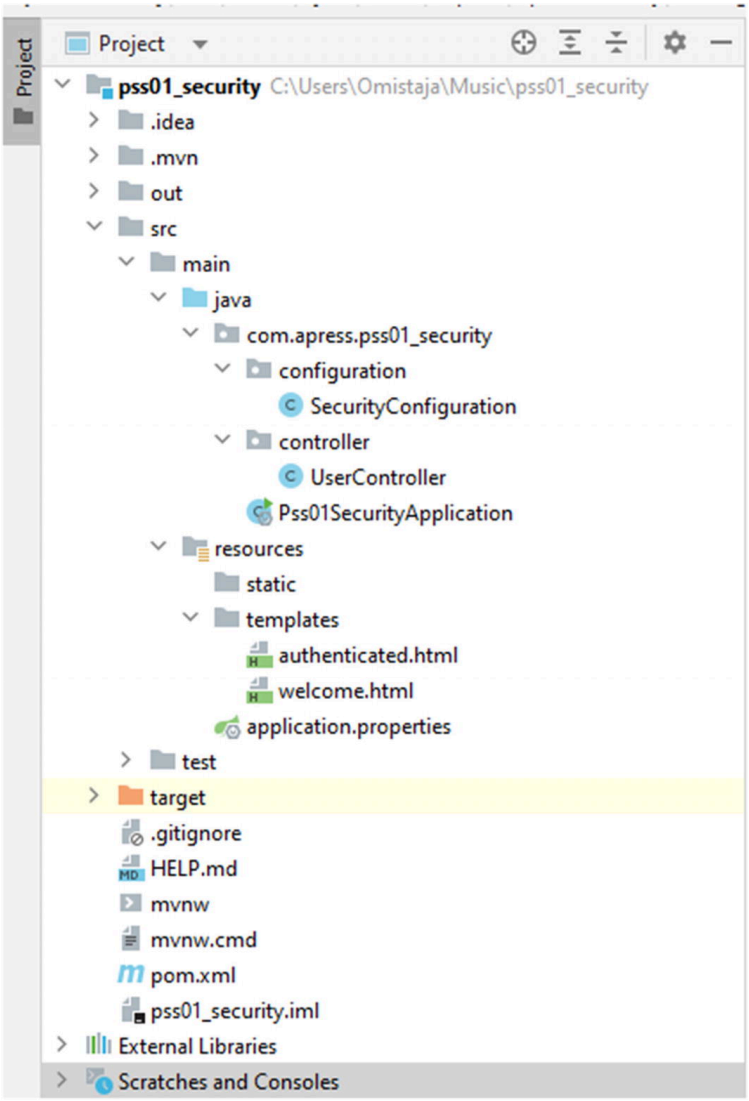

Figure 5-6 New Spring Security 6 in Boot 3 project structure

Next, build and run the Spring Security 6 project.

You can now build the project, deploy the JAR file, start the application running on the stand-alone Tomcat Server 10, and deploy the JAR file automatically.

Your application is deployed successfully. Open the web browser and type the following link: `http://localhost:8080/welcome/`. The outcome is shown in Figure 5-7.

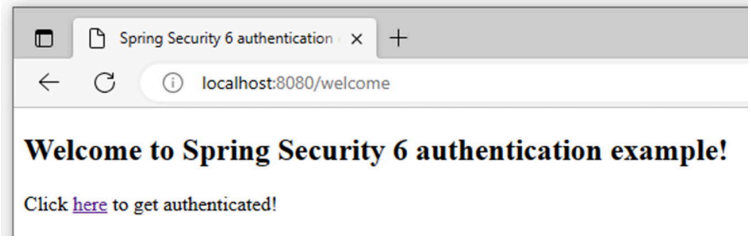

Figure 5-7 Browsing the new Spring Security project

You can now access the security `authenticated.html` by clicking the Login link. The outcome is shown in Figure 5-8.

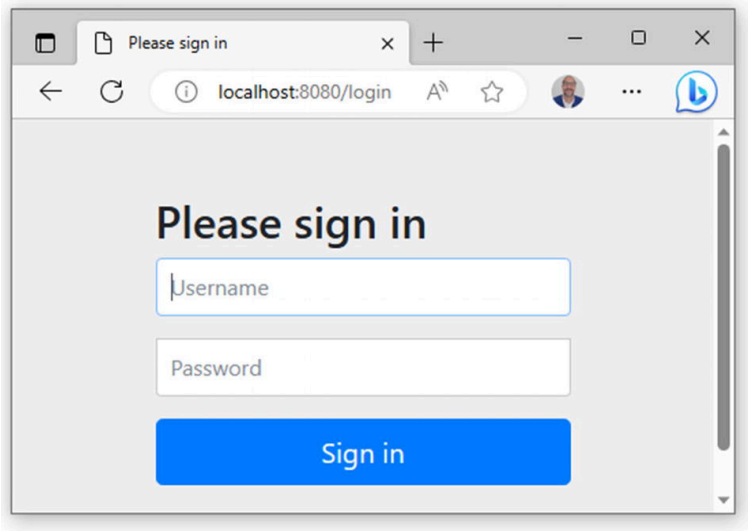

Figure 5-8 Accessing the Spring Security login web page

Now, if you access with bad credentials, such as without having the "ADMIN" role, you receive an error message like the one in Figure 5-9.

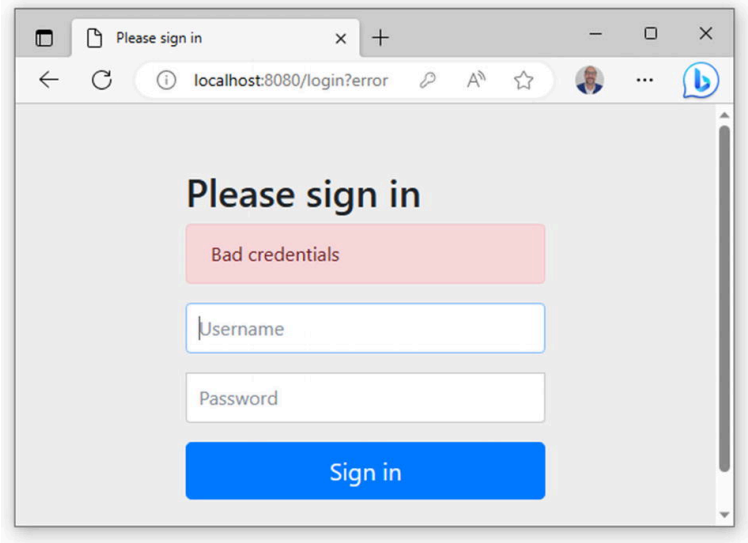

Figure 5-9 Accessing with wrong login credentials

As you can see, Spring Security directly produces the login error and reminds the user that the credentials provided are incorrect.

If you next provide the right user admin/adminpassw credentials for the "Admin" role, you will receive the content defined in the `authenticated.html` page, as shown in Figure 5-10.

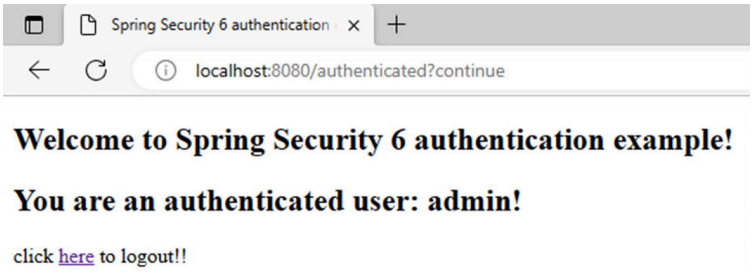

Figure 5-10 Accessing with the right admin credentials

If you log out, you see the result shown in Figure 5-11 and are redirected to the Welcome page.

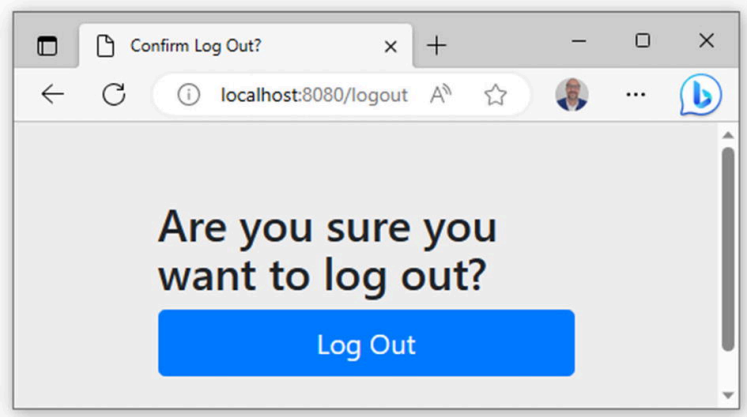

Figure 5-11 Logout page

When you make the HTTP request to the configured URL, and after your servlet container deals with it, the request lands in the `DelegatingFilterProxy`, which delegates the processing to the security `FilterChainProxy`.

In general, Spring Security utilizes a lot of filters. The HTTP request filter is used to do the following.

- Intercept the request.
- Detect authentication (or absence of).
- Redirect to the authentication entry point.
- Pass the request to the authorization service.
- Send the request to the servlet or throw a security exception.

In Spring Security 6, the most important filters are the following.

- ForceEagerSessionCreationFilter
- ChannelProcessingFilter
- WebAsyncManagerIntegrationFilter
- SecurityContextPersistenceFilter
- HeaderWriterFilter
- CorsFilter
- CsrfFilter
- LogoutFilter
- OAuth2AuthorizationRequestRedirectFilter
- Saml2WebSsoAuthenticationRequestFilter
- X509AuthenticationFilter
- AbstractPreAuthenticatedProcessingFilter
- CasAuthenticationFilter
- OAuth2LoginAuthenticationFilter
- Saml2WebSsoAuthenticationFilter
- UsernamePasswordAuthenticationFilter
- DefaultLoginPageGeneratingFilter
- DefaultLogoutPageGeneratingFilter
- ConcurrentSessionFilter
- DigestAuthenticationFilter

- BearerTokenAuthenticationFilter
- BasicAuthenticationFilter
- RequestCacheAwareFilter
- SecurityContextHolderAwareRequestFilter
- JaasApiIntegrationFilter
- RememberMeAuthenticationFilter
- AnonymousAuthenticationFilter
- OAuth2AuthorizationCodeGrantFilter
- SessionManagementFilter
- ExceptionTranslationFilter
- AuthorizationFilter
- SwitchUserFilter

The following are the most important Spring Security 6 filters.

- BasicAuthenticationFilter tries to authenticate the user with the header's username and password if it finds a Basic Auth HTTP header on the request.
- UsernamePasswordAuthenticationFilter tries to authenticate the user with those values if it finds a username/password request parameter/POST body.
- DefaultLoginPageGeneratingFilter generates a default login page when enabling Spring Security unless you don't explicitly disable that feature.
- DefaultLogoutPageGeneratingFilter generates a logout page unless you explicitly disable that feature.
- FilterSecurityInterceptor does the authorization.

Let's learn more about our example's Spring Security 6 filters.

The HTTP request and authentication processes and filters are explained in Chapter 4.

Let's see what happens when incorrect or correct credentials are provided when logging in. When the browser is redirecting and asks for `/login`, the following occurs, the process is the same as the first request until it reaches the `DefaultLoginPageGeneratingFilter`. At this point, the filter detects the request for `/login` and writes the login form's HTML data directly in the response object. Then, the response is rendered.

Now, try to log in with incorrect credentials. Let's follow the request through the framework to see what happens.

1. In the login form, type **admin** as the username and **adminpassw** as the password.
2. When the form is submitted, the filters are activated again in the same order. This time, however, when the request arrives at the `UsernamePasswordAuthenticationFilter`, the filter checks whether the request is for `/login` and sees that this is indeed the case. The filter extracts the username and password authentication information from the HTTP request parameters `username` and `password`, respectively. With this information, it creates the `UsernamePasswordAuthenticationToken Authentication` object,

which then sends it to the `AuthenticationManager` (or, more exactly, its default implementation, `ProviderManager`) for authentication.

3. `DaoAuthenticationProvider` is called from the `ProviderManager` with the `Authentication` object. The `DaoAuthentication` provider implements `AuthenticationProvider`, which uses a strategy of `UserDetailsService` to retrieve the users from whichever storage they live in. With your current configuration, it tries to find the username of the configured `InMemoryUserDetailsManager` (the implementation of `UserDetailsService` that maintains an in-memory user storage in a `java.util.Map`). Because no user has this username, the provider throws a `UsernameNotFoundException` exception.
4. The provider catches this exception and converts it into a `BadCredentialsException` to hide the fact that there is no such user in the application; instead, it treats the error as a common username-password combination error.
5. `UsernamePasswordAuthenticationFilter` catches the exception. This filter delegates to an instance of an implementation of `AuthenticationFailureHandler`, which in turn decides to redirect the response to `/login?error`. This way, the login form is displayed again in the browser with an error message.

Documentation on filters is at **https://docs.spring.io/spring-security/reference/6.1-SNAPSHOT/servlet/architecture.html#servlet-filters-review**.

Restart the application and go to `http:/localhost:8080/welcome`, which triggers the login page. Type **admin** as the username and **adminpassw** as the password in the form. Then click the Login button.

- The request follows the same filter journey as before. This time, `InMemoryUserDetailsManager` finds a user with the requested username and returns that to `DaoAuthenticationProvider`, which creates a successful `Authentication` object.
- After successful authentication, `UsernamePasswordAuthenticationFilter` delegates to an instance of `SavedRequestAwareAuthenticationSuccessHandler`, which looks for the original requested URL (`/authenticated`) in the session and redirects the response to that URL.

When `http://localhost:8080/authenticated` is requested, the request works through the filter chain as in the previous cases. This time, though, you already have a fully authenticated entity in the system. The request arrives in `FilterSecurityInterceptor`.

- `FilterSecurityInterceptor` receives an access request to `/ authenticated`. Then, it recovers the necessary credentials to access that URL (`ROLE_ADMIN`).
- The `AffirmativeBased` access-decision manager gets called and calls the `RoleVoter` voter. The voter evaluates the authenticated entity's authorities and compares them with the required credentials to access the resource. Because the voter finds a match (`ROLE_ADMIN` is

in both the `Authentication` authorities and the resource's `config` attributes), it votes with an `ACCESS_GRANTED` vote.

- `FilterSecurityInterceptor` forwards the request to the next element in the request-handling chain, which, in this case, is Spring's `DispatcherServlet`.
- The request gets to the `AdminController`, which simply returns the authenticated page .
- This is the complete flow of the authentication and authorization process. Figure 5-12 shows this full interaction in a pseudo flow chart.

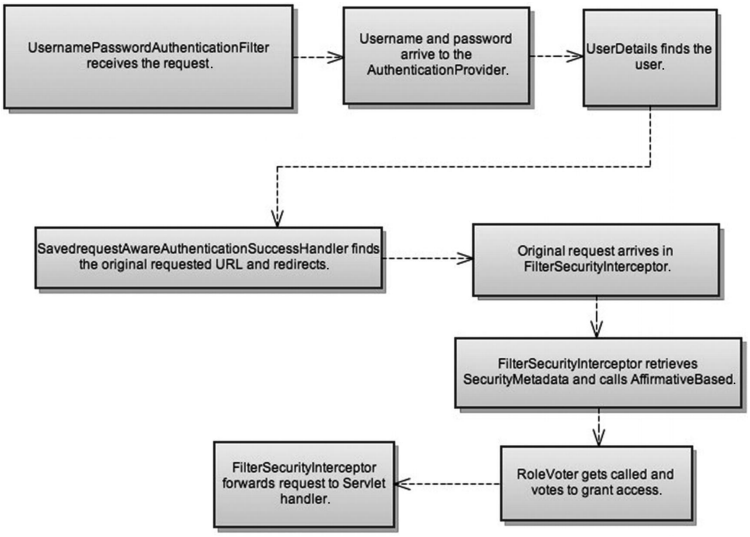

Figure 5-12 Overall flow of a successful authentication and authorization process

The Special URLs

From the preceding explanation, you can see that Spring Security's support for web security defines a few preconfigured URLs for you to use in your application. These URLs get special treatment in the framework. The following are the main ones.

- `/login`: This is the URL that Spring Security uses to show the login form for the application. The framework redirects to this URL when an authentication is needed, but it doesn't exist yet.
- `/logout`: The framework uses this URL to log out the currently logged-in user, invalidating the corresponding session and SecurityContext.

In the previous URLs, the first thing that comes to mind is how to configure your own login form in the application and, in general, how to customize the login process instead of using the default one. That is what we'll do next.

Note `/login` replaced `/j_spring_security_check` in Spring Security 5.

Custom Login Form

The user authentication request to your application has been made via the `http.authorizeRequests()` method since Spring Security 5.

When you configure the `http` element via the `http.authorizeHttpRequests ()` method, as you did before, Spring

Security sets up a default login and logout process for you, including a login URL, login form, default URL after login, and other options. Basically, when Spring Security's context starts to load up, it finds that there is no custom login page URL configured, so it assumes the default one and creates a new instance of `DefaultLoginPageGeneratingFilter` that is added to the filter chain. This filter is the one that generates the login form for you.

If you want to configure your own form, you must do a few tasks. First, tell the framework to replace the default handling with your own. You define the following element as a child of the `http.authorizeRequests()` method in the `SecurityConfiguration` Java file.

```
formLogin((form) -> form
```

This element tells Spring Security to change its default login-handling mechanism on startup. `DefaultLoginPageGeneratingFilter` is no longer instantiated. Let's try the first configuration. With the new configuration in place, restart the application and try to access `http://localhost:8080/ /authenticated`.

You are redirected to `/login` and get a 404 HTTP error because you haven't defined any handler for this URL yet, as shown in Figure 5-13.

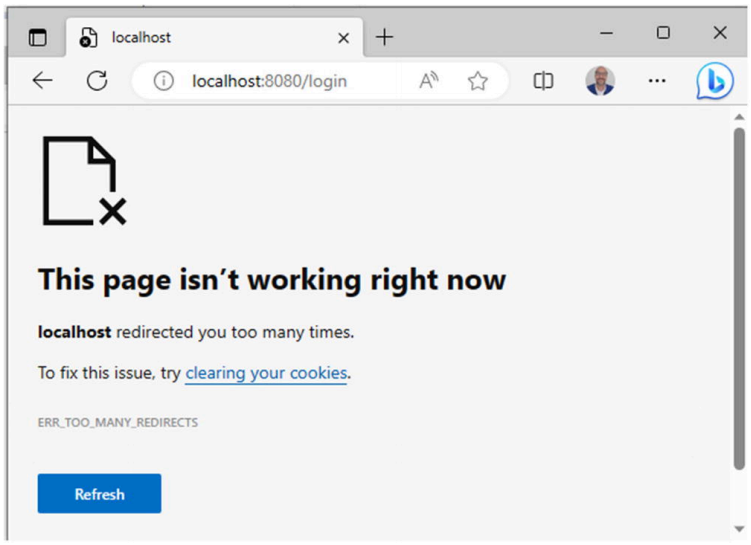

Figure 5-13 Error 404 that appears when defining a new login handler page

Let's add a login controller in the `UserController`, as shown in Listing 5-7.

```
  @GetMapping("/login")

    public String loginPage() {

       return "login";

  }
```

Listing 5-7
Login Controller Added to the UserController

Next, add the following line to the SecurityConfiguration file.

```
formLogin((form) -> form

        .loginPage("/login")

        .permitAll()

)
```

Create the `login.html` page from Listing 5-8 in the `templates` folder in your application.

```
<!DOCTYPE html>

<html xmlns:="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org">

<head>

  <title>Spring Security Example </title>
```

```
</head>

<body>

<div th:if="${param.error}">

  Invalid username and password.

</div>

<div th:if="${param.logout}">

  You have been logged out.

</div>

<h1>Spring Security v6 Custom Login Form</h1>

<h2>Login with Username and Password:</h2>

<form th:action="@{/login}" method="post">

  <div><label> Username : <input type="text" name="username"/> </label></div>

  <div><label> Password: <input type="password" name="password"/> </label></div>

  <div><label>Remember Me:<input type="checkbox" name="remember-me"/> </label></div>

  <div><input type="submit" value="Login"/></div>

</form>

</body>

</html>
```

Listing 5-8

Custom login.html

In the authenticated.html file, replace the following line

```
<p>Click <a th:href="@{/logout}">here</a> to logout!!</p>
```

with this

```
<form th:action="@{/logout}" method="post">
    <input type="submit" value="Logout"/>
</form>
```

If you restart the application and again go to `http://localhost:8080/authenticated`, you should see your new login form when you get redirected to the `/login` URL. The form is shown in Figures 5-14 and 5-15. If you type **admin** as the username and **adminpassw** as the password, you can access the `authenticated` page, as you did with the default login form.

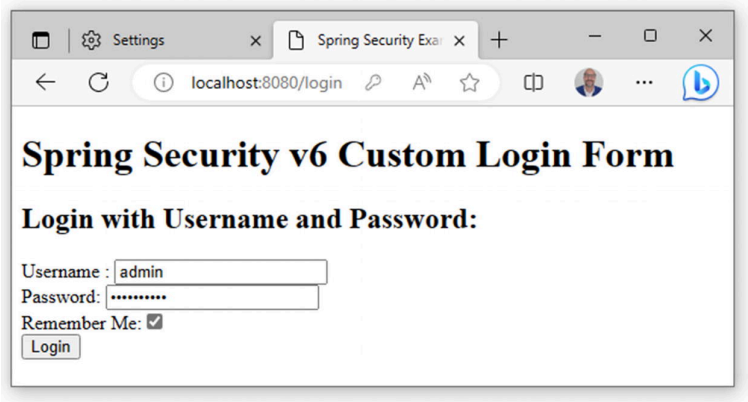

Figure 5-14 Custom login form

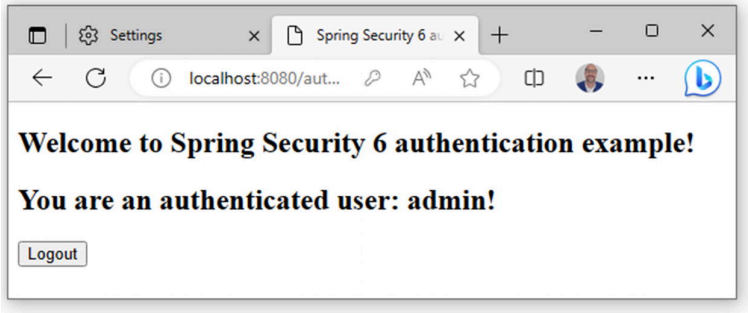

Figure 5-15 Successful custom login form

Click the Logout button to sign out the current user.

If you look at the login.html, you can see certain names for the username field, password field, the remember me checkbox, and the form element's action attribute. These are not random names. Spring Security expects using these particular names to treat the authentication process correctly. Also, the form should use POST to send the information to the server because the framework requires this.

The Remember Me checkbox shown in Figure 5-16 is explained later.

The element `<form-login>` supports many more configuration options, including changing the authentication request parameters' default username and password names.

The `<form-login>` attributes might include

- `always-use-default-target`
- `authentication-details-source-ref`
- `authentication-failure-handler-ref`
- `authentication-failure-url`
- `authentication-success-handler-ref`
- `default-target-url`
- `login-page`
- `login-processing-url`
- `password-parameter`
- `username-parameter`
- `authentication-success-forward-url`
- `authentication-failure-forward-url`

Give this attribute the value `/login`. Then, in your `login.html`, add the content from Listing 5-9 just after the `<body>` tag.

```
<div th:if="${param.error}">
  Invalid username and password.
</div>
```

Listing 5-9

Snippet Showing an Error in login.jsp

If you restart the application and try to access `http://localhost:8080//authenticated` and use an incorrect username and password, you are sent to the login page again, but with an "Invalid username and password." error message shown at the top, as shown in Figure 5-16.

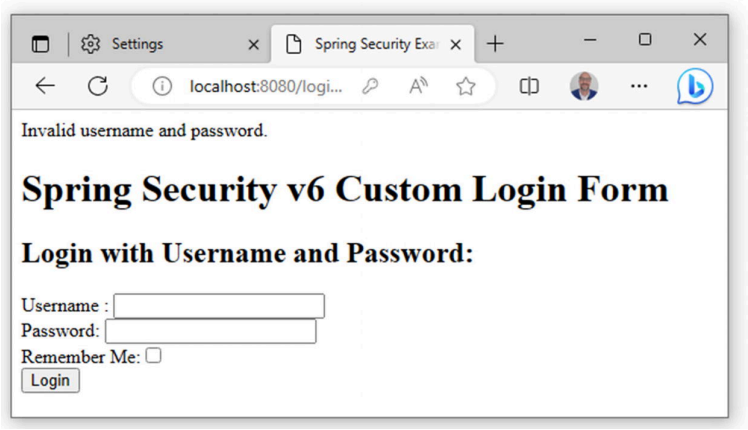

Figure 5-16 A custom error shown in the custom form

Note that this URL could be a different one, unrelated to the login URL. But the common pattern is to allow the user another attempt at logging in, showing her any errors.

- `authentication-success-handler-ref`: Reference to an `AuthenticationSuccessHandler` bean in the Spring application context. This bean is called upon successful authentication and should handle the next step after authentication, usually deciding the redirect destination in the application. A current implementation in the form of `SavedRequestAwareAuthenticationSuccessHandler` takes care of redirecting the logged-in user to the original requested URL after successful authentication.
- `authentication-failure-handler-ref`: Reference to an `AuthenticationFailureHandler` bean in the Spring application context. It is used to handle failed authentication requests. When an authentication fails, this handler gets called. A standard behavior for this handler is to present the login screen again or return a 401 HTTP status error. This behavior is provided by the concrete class `SimpleUrlAuthenticationFailureHandler`.

When authenticating a Spring Security application, there are three different interfaces to consider: AuthenticationSuccessHandler, AuthenticationFailureHandler, and AccessDeniedHandler.

Let's develop a simple example implementation of the `AuthenticationFailureHandler interface`. It returns a 500 status code

when failing to authenticate. Create the class `CustomAuthenticationFailureHandler` from Listing 5-10.

```
package com.apress.pss01_security.configuration;



import jakarta.servlet.http.HttpServletRequest;

import jakarta.servlet.http.HttpServletResponse;

import org.springframework.security.core.AuthenticationException;

import org.springframework.security.web.authentication.AuthenticationFailureHandler;



import java.io.IOException;



public class CustomAuthenticationFailureHandler implements AuthenticationFailureHandler {



    @Override

    public void onAuthenticationFailure(HttpServletRequest request, HttpServletResponse response, Authe

            throws IOException {

        response.sendError(500);

    }

}
```

Listing 5-10

AuthenticationFailureHandler Implementation for ServerErrorFailureHandler

Add the following to the `SecurityConfiguration` class file.

```
.formLogin((form) -> form

        .loginPage("/login")

        .defaultSuccessUrl("/authenticated")

        .permitAll()

        .failureHandler(authenticationFailureHandler())
```

Add the following to the new bean.

```
@Bean

public AuthenticationFailureHandler authenticationFailureHandler() {

    return new CustomAuthenticationFailureHandler();

}
```

Restart the application, go to `http://localhost:8080/authenticated`, use a random username and password, and click the Submit button. You should get a 500 error in the browser.

Basic HTTP Authentication

Sometimes, you can't use a login form for authenticating users. For instance, if your application is meant to be called by other systems instead of a human user, showing a login form to the other application doesn't make sense. This is a pretty common use case. Web services talk to each other without user interaction, ESB systems integrate with one another, and JMS clients produce and consume messages from other systems.

In the context of HTTP-exposed interfaces that require no human user to access them, a common approach is to use HTTP Basic authentication headers. HTTP authentication headers allow you to embed the security information (username and password) in the header of the request that you send to the server instead of sending it in the body of the request, as is the case for the login form authentication.

HTTP uses a standard header for carrying this information. The header is appropriately named `Authorization`. When using this header, the client that is sending the request (for example, a browser) concatenates the username and the password with a colon between them. Then, Base64 encodes the resulting string, sending the result in the header. For example, if you use **neve** as the username and **nardone** as the password, the client creates a `neve:nardone` string and encodes it prior to sending it in the header.

Let's use basic HTTP authentication in your application. The first and only thing you need to do is remove any authentication method in your `SecurityConfiguration` file and instead add the following.

```
.httpBasic(withDefaults())
```

After replacing it, restart the application and go to `http://localhost:8080/authenticated` in the browser. A standard HTTP authentication pop-up box asks for your authentication details, as Figure 5-17 shows. Type **admin** and **admin123** as the username and password, and send the request. You will successfully arrive on the movies page (see Figure 5-5).

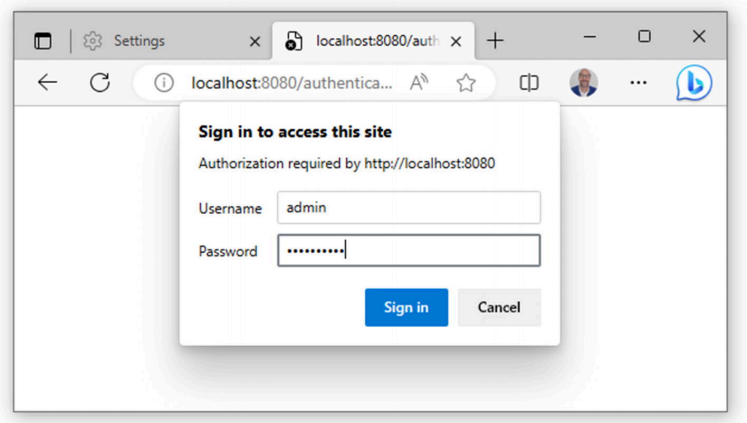

Figure 5-17 Standard HTTP authentication form, Basic authentication configuration

When you use the `httpBasic` configuration element, Spring `Security's BasicAuthenticationFilter` appears. A `BasicAuthenticationEntryPoint` strategy is configured into the `ExceptionTranslationFilter` on startup. When you make the first request to `/movies`, the framework behaves as before, throwing an access-denied exception that the `ExceptionTranslationFilter` handles. This filter delegates to a particular implementation strategy of `AuthenticationEntryPoint`—in this case, `BasicAuthenticationEntryPoint`. `BasicAuthenticationEntryPoint` adds the header `WWW-Authenticate: Basic realm="Spring Security Application"` to the response and then sends the client an HTTP status of 401 (Unauthorized). The client should know how to handle this code and work accordingly. (In the case of a browser, it simply shows the authentication pop-up.)

When you introduce the username and password and submit the request, the request again follows the filter chain until it reaches the `BasicAuthenticationFilter`. This filter checks the request headers, looking for the `Authorization` header starting with `Basic`. The filter extracts the content of the header and uses `Base64.decode` to decode the string, then extracts the username and password. The filter creates a `UsernamePasswordAuthenticationToken` object and sends it to the authentication manager for authentication in the standard way. The authentication manager asks the authentication provider to retrieve the user and create an `Authentication` object. This process is standard and independent of using Basic authentication or form authentication.

Digest Authentication

Digest authentication helps to solve many of the weaknesses of Basic authentication, specifically by ensuring credentials are never sent in clear text across the wire.

Digest authentication is a very close sibling of basic HTTP authentication. Its main purpose is to avoid sending clear text passwords on the wire, as Basic authentication does, by hashing the password before sending it to the server. This makes Digest authentication more complex than Basic authentication.

Digest authentication works with HTTP headers in the same way that Basic authentication does.

Digest authentication is based on using a nonce for hashing the passwords. A *nonce* is an arbitrary server-generated number that is used in the authentication process and is used only once. It is passed through the digest computation with the username, password, nonce, URI being requested, and so on.

In the authentication process, the server and client do the digest computation, which should match.

A *nonce* is central to Digest authentication. It is a value the server generates. The following shows Spring Security's nonce format.

```
Digest Syntax

base64(expirationTime + ":" + md5Hex(expirationTime + ":" + key))

expirationTime:   The date and time when the nonce expires, expressed in milliseconds

key:              A private key to prevent modification of the nonce token
```

The main processing lies in two classes: `DigestAuthenticationFilter` and `DigestAuthenticationEntryPoint`.

`DigestAuthenticationFilter` queries the request's headers, looking for the Authorization header, and then it checks that the header's value starts with `Digest`. If this is the case, the request carries the security credentials used for authentication.

`DigestAuthenticationEntryPoint` is the class invoked to generate a response that demands a digest security authentication process begin. This class sets the header `WWW-Authenticate` with the correct values (including the nonce) so that the client agent (the browser) knows it has to start the Digest authentication process.

To configure the Digest authentication, update the `SecurityConfiguration` file with the following lines.

```
.exceptionHandling(e -> e.authenticationEntryPoint(authenticationEntryPoint()))
```

```
.addFilterBefore(digestFilter());
```

You must ensure that you configure insecure plain text Password Storage using NoOpPasswordEncoder.

Next, add the following bean to configure the Digest authentication.

```
@Autowired
UserDetailsService userDetailsService;

DigestAuthenticationEntryPoint entryPoint() {
      DigestAuthenticationEntryPoint result = new DigestAuthenticationEntryPoint();
      result.setRealmName("My Security App Realm");
      result.setKey("3028472b-da34-4501-bfd8-a355c42bdf92");
}

DigestAuthenticationFilter digestAuthenticationFilter() {
      DigestAuthenticationFilter result = new DigestAuthenticationFilter();
      result.setUserDetailsService(userDetailsService);
      result.setAuthenticationEntryPoint(entryPoint());
}
```

Then, define the username and password using `inMemoryUserDetailsManager`.

```
    @Override
    @Bean
    public UserDetailsService userDetailsServiceBean() {
        InMemoryUserDetailsManager inMemoryUserDetailsManager = new InMemoryUserDetailsManager();
        inMemoryUserDetailsManager.createUser(User.withUsername("admin").password(passwordEncoder.encod
        return inMemoryUserDetailsManager;
    }
```

If you restart the application and go to `http://localhost:8080/authenticated`, you are presented with a browser dialog box asking for a username and password exactly like the one shown for Basic authentication. This is the `DigestAuthenticationEntryPoint`'s work. As explained, the entry point fills the response object with the required headers so that the browser knows it needs to show the login form. Log in with the **admin** username and **adminpassw** as the password, and you should be able to access the requested URL.

The browser creates its own digested message with the password input included and puts it in the header. It also puts the rest of the information—namely, nonce, cnonce, realm, and so on—in the `Digest` header. The following is a `Digest` header sent to the server with your current request.

```
'Digest username="admin", realm=" Security Digest Authentication",
```

```
nonce="MTM1NTY3NDc3NDIy....==", uri=" /authenticated",

response="225ea6fbad618cfdf1da7d4f7efe53b8", qop=auth,

nc=00000002, cnonce="376a9b27621880bd"'
```

When the request reaches `DigestAuthenticationFilter`, the request headers contain the required Digest authentication header. The information in this header arrives as a CSV string containing all the required information shown in the last paragraph, including the nonce and the client nonce (`cnonce`). (A nonce is an arbitrary number used only once in a cryptographic communication. See **http://en.wikipedia.org/wiki/Cryptographic_nonce**.). The filter extracts the information from the header, retrieves the user from the `UserDetailsService`, and then computes the digest with the password from the retrieved user to see if it matches the one sent in the header by the client. If they match, access is granted.

Remember-Me Authentication

The remember-me authentication functionality allows returning application users to use it without logging in every time. The application remembers certain visitors, allowing them to just open the application and be greeted with their personalized version of the application as if they were logged in.

Remember-me functionality is very convenient for users but is also very dangerous and recommended for private (from home) use only.

The problem should be obvious. If you use an application from a public computer and this application remembers your profile information, the next person who accesses that application from that computer can impersonate you with minimum effort.

It is also common practice to offer limited functionality in the remember-me session. This means that even if you are logged in automatically, thanks to the remember-me functionality, you won't have access to the whole functionality of the application. More sensitive parts of the application might require you to formally log in to use them.

This is the case, for example, with Amazon.com. When you visit Amazon.com and log in, the next time you visit Amazon, the site remembers you, your recommendations, your name, and other information about you. But to buy something, you must log in fully to access that functionality.

Remember-me authentication is typically supported by sending a cookie to the browser, which then, on subsequent sessions in the application, is sent back to the server for auto login.

How does the remember-me functionality work in Spring Security?

Remember-me functionality in Spring Security is mainly supported by the `RememberMeServices` interface and the `RememberMeAuthenticationFilter` class. Let's see how they work in the context of a request.

When the application starts, the `RememberMeAuthenticationFilter` is in the server's filter chain. Also, a `TokenBasedRememberMeServices` is instantiated and injected into the `AbstractAuthenticationProcessingFilter`, replacing the no-op `NullRememberMeServices`.

Go to `http://localhost:8080/authenticated`, and log in with **admin** as the username and **adminpassw** as the password.

When the request gets into the application, `UsernamePasswordAuthenticationFilter` (a subclass of `AbstractAuthenticationProcessingFilter`) handles the authentication process in the standard way already explained.

After successful authentication, `UsernamePasswordAuthenticationFilter` invokes the configured `TokenBasedRememberMeServices`'s `loginSuccess` method. This method looks to see if the request contains the `parameter remember-me` to apply the remember-me functionality. (If the property `alwaysRemember` is set to `true` in the service, it also applies the remember-me functionality.) Because you didn't send this request, nothing happened.

So let's add the parameter to the login form you have. Open the `login.html` file, and paste the following element somewhere inside `<form>`.

```
<div><label>Remember Me:<input type="checkbox" name="remember-me"/> </label></div>
```

In the `SecurityConfiguration` configuration file, add the following.

```
.rememberMe((remember) -> remember

        .rememberMeParameter("remember-me")
```

```
            .key("uniqueAndSecretKey")

            .tokenValiditySeconds(1000)

            .rememberMeCookieName("rememberloginnardone")

            .rememberMeParameter("remember-me")

    )
```

These lines define the key name, the parameter name, the cookie name, and the validity time in seconds.

Restart the application and visit `http://localhost:8080/authenticated`. You should now see a check box with username and password fields. Select the check box, and log in with **`admin`**/**`adminpassw`**.

This time, the request carries the required parameter, and `TokenBasedRememberMeServices` works. It extracts the username and password from the `Authentication` object and creates a token with this information and an expiration time. And it creates an MD5 encoding out of the resulting string. This value is then Base65-encoded with the username again and added to the response as a cookie named rememberloginnardone that is returned to the browser. You can see this cookie in Figure 5-18.

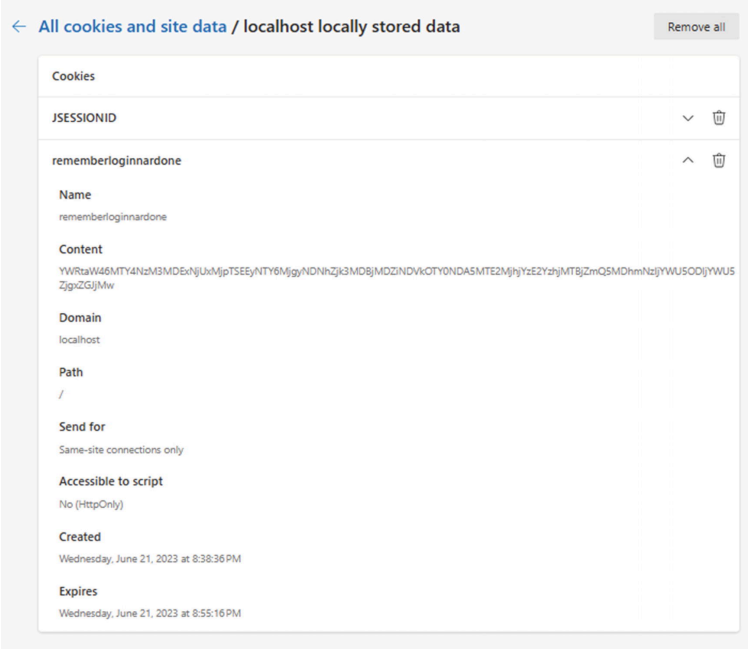

Figure 5-18 Remember-me cookie example

Restart the application. Visit `http://localhost:8080/authenticated`. You should be able to access the page without logging in.

When this request gets into the system, it is intercepted by `RememberMeAuthenticationFilter`, which goes into action. The first thing the filter does is check that there is no current `Authentication` in `SecurityContext`. Because this means there is no user logged in, the filter calls `RememberMeServices`'s `autoLogin` method.

In the standard configuration, `TokenBasedRememberMeServices` is the concrete class that implements `RememberMeServices`. This implementation's `autoLogin` method tries to parse the incoming cookie into its composing elements, which are the username, the hashed value of the combined elements (

base64(username + ":" + expirationTime + ":" + algorithmName + ":"

algorithmHex(username + ":" + expirationTime + ":" password + ":" + key))), and the expiry time of the token. Then, it retrieves the `UserDetails` from the `UserDetailsService` with the username, recomputes the hashed value with the retrieved user, and compares it with the arriving one. If they don't match, an `InvalidCookieException` is thrown. If they do match, `UserDetails` is checked, and an `Authentication` object is created and returned to the caller.

The `autoLogin` method extracts the remember-me cookie from the request, decodes it, does some validation, and then calls the configured `UserDetailsService`'s `loadUserByUsername` method with the username extracted from the cookie. It then creates a `RememberMeAuthenticationToken` object (an implementation of `Authentication`).

The `RememberMeAuthenticationFilter` then tries to authenticate this new `Authentication` object against the `AuthenticationProvider`'s implementation of `RememberMeAuthenticationProvider`, which simply returns the same `Authentication` object after making sure that the hash from the incoming request matches the stored one for the remember-me key.

The security interceptor uses this `Authentication` object to allow access to the requested URL.

Logging Out

Logging out is pretty simple. When you log out of an application, you want the application to end your current session and remove any information it might have stored on the client for you.

`/logout` has replaced `/j_spring_security_logout` since Spring Security 5.

In Spring Security, logging out is very easy. The only thing you need to do by default is to visit `/logout`. Let's try that.

Add the following lines to the `UserController` file.

```
@GetMapping ("/logout")
public String logoutPage() {
    return "redirect:/welcome";
}
```

Update the `SecurityConfiguration` file with the following lines.

```
.logout((logout) -> logout
        .logoutSuccessUrl("/welcome")
        .deleteCookies("JSESSIONID")
        .invalidateHttpSession(true)
        .permitAll()
);
```

```
.logout((logout) -> logout
        .logoutSuccessUrl("/welcome")
        .deleteCookies("JSESSIONID")
        .invalidateHttpSession(true)
        .permitAll()
```

```
);
```

These lines tell the application to delete the JSESSIONID cookie, invalidate the HTTP session, and redirect to the index web page once logged out.

Now go to `http://localhost:8080/aithenticated` and log in with **admin**/**adminpassw** again. Select the check box for activating `remember-me` functionality. You should be able to log in without problems.

If you look at the cookies stored in your browser, you should see two cookies for the localhost domain: `JSESSIONID and rememberloginnar-done`. Figure 5-18 shows the two cookies. If you log out, you would expect these two cookies to disappear from the browser, basically removing any trace of the application from your browser. Let's do it.

Click the logout link on the `movies.jsp` page. You should be logged out of the application. If you open your browser's cookies, you see that the cookie `rememberloginnardone` is gone. The `JSESSIONID` cookie exists, but the framework already invalidated the session. Figure 5-19 shows remember-me and session cookies.

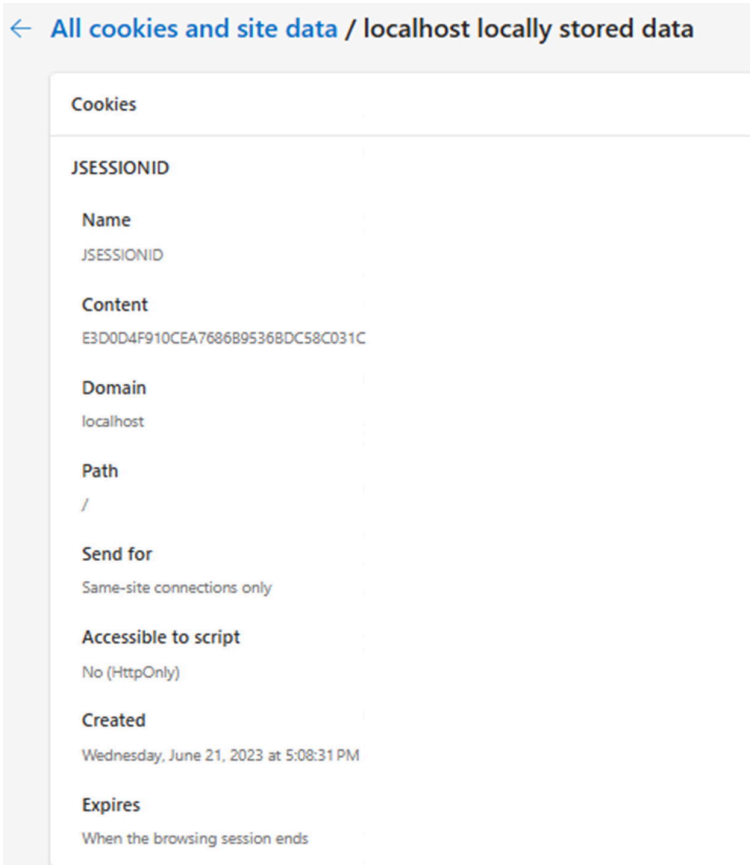

Figure 5-19 Remember-me and session cookies

The flow of the logout request is as follows: When the request arrives, it follows the filter chain until it arrives at `LogoutFilter`. This filter notices

that the URL that is being requested is for logout. The filter calls the configured `LogoutHandler(s)`, which in the running application are `SecurityContextLogoutHandler` and `TokenBasedRememberMeServices`. (They implement the `LogoutHandler` interface.)

`SecurityContextLogoutHandler` invalidates the servlet session in the standard servlet way, calling the `invalidate` method on the `HttpSession` object and clearing the `SecurityContext` from Spring Security as shown using

```
.logout()

.invalidateHttpSession(true).
```

`TokenBasedRememberMeServices` simply removes the remember-me cookie by setting its age to 0.

Finally, the JSESSIONID cookie was deleted by adding the `.logout()` line.

```
.deleteCookies("JSESSIONID")
```

Session Management

Another area of Spring Security's web support is the management of user sessions. One very important thing to do regarding sessions is to create a new session ID when a user authenticates successfully. Doing this reduces the likelihood of session fixation attacks, in which one user sets another user's session identifier to impersonate him in the application. Spring Security also offers a feature to specify the number of concurrent sessions the same user can have open at any given time.

These two features are controlled by the `SessionFixationProtectionStrategy` class, which implements `SessionAuthenticationStrategy`. This strategy is invoked from `AbstractAuthenticationProcessingFilter` and `SessionManagementFilter`. Let's look at how they work.

`SessionFixationProtectionStrategy` is already configured by default in `UsernamePasswordAuthenticationFilter`, which is configured in the application. When you log in, this strategy is invoked. When the strategy is invoked, it retrieves the current session (normally anonymous) and invalidates it. It immediately creates a new one. It also tries to migrate certain attributes—normally, those used by Spring Security, but a list can also be specified.

To summarize this strategy, when you log in, it invalidates the current session, creates a new one, and copies certain attributes from the old one to the new one.

Since Spring Session 2.0 contains the Spring Session Core module and several other modules like Spring Session Data MongoDB module, Spring Session Data Geode modules, etc.

In Spring Security, you can control exactly when our session is created and how to interact with it by defining the following line to the `SecurityConfiguration` class.

```
.sessionCreationPolicy(SessionCreationPolicy."ADD A VALUE")
```

The value can be one of the following.

- always: A session is always created if one doesn't already exist.
- ifRequired: A session is created if required (default).
- never: The framework will never create a session itself, but it will use one if it already exists.
- stateless: No session will be created or used by Spring Security.

The first step in enabling this feature is adding the HttpSessionEventPublisher listener to your application to limit multiple logins for the same user through session management.

```
@Bean
public HttpSessionEventPublisher httpSessionEventPublisher() {
    return new HttpSessionEventPublisher();
}
```

Let's go over how to configure it in your application.

1. Add the following line to the `SecurityConfiguration` file.

```
.sessionCreationPolicy(SessionCreationPolicy.ALWAYS)
 .maximumSessions(1))
```

`.maximumSessions(1)`) means that no multiple concurrent sessions are possible.

2. Restart the application.
3. Open Chrome and go to `http://127.0.0.1:8080/authenticated`. Log in with username **admin** and password **adminpassw**. You should be able to access the page without a problem.
4. Open another browser, for instance, Firefox, and visit `http://127.0.0.1:8080/authenticated`. Log in with the **admin** username and the **adminpassw** password. You should be able to access the page without a problem.
5. Go to Chrome and refresh the page. You get the message: "This session has expired (possibly due to multiple concurrent logins being attempted as the same user)."

Now let's allow two sessions at the same time by adding the following line to the `SecurityConfiguration` file.

```
.sessionManagement().maximumSessions(2)
```

Restart the application and follow the same flow as before. This time, you should have both sessions active at the same time.

Finally, in Listing 5-11, you see the entire `SecurityConfiguration` Java class.

```
package com.apress.pss01_security.configuration;



import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.Customizer;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.http.SessionCreationPolicy;

import org.springframework.security.core.userdetails.User;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.security.provisioning.InMemoryUserDetailsManager;

import org.springframework.security.web.SecurityFilterChain;

import org.springframework.security.web.access.AccessDeniedHandler;

import org.springframework.security.web.authentication.AuthenticationFailureHandler;

import org.springframework.security.web.authentication.AuthenticationSuccessHandler;

import org.springframework.security.web.authentication.www.DigestAuthenticationEntryPoint;
```

```
import org.springframework.security.web.authentication.www.DigestAuthenticationFilter;

import org.springframework.security.web.session.HttpSessionEventPublisher;



@Configuration

@EnableWebSecurity



public class SecurityConfiguration {



    @Bean

    public SecurityFilterChain filterChain1(HttpSecurity http) throws Exception {

        http

                .authorizeHttpRequests((authorize) -> authorize

                        .requestMatchers("/", "/welcome").permitAll()

                        .requestMatchers("/authenticated").hasRole("ADMIN")

                        .requestMatchers("/customError").permitAll()

                        .anyRequest().denyAll()

                )



                .csrf(Customizer.withDefaults())
```

```
                // .httpBasic(withDefaults())    using Basic Authentication

                //.formLogin(withDefaults())     using Form Authentication not customized



                .rememberMe((remember) -> remember

                        .rememberMeParameter("remember-me")

                        .key("uniqueAndSecretKey")

                        .tokenValiditySeconds(1000)

                        .rememberMeCookieName("rememberloginnardone")

                        .rememberMeParameter("remember-me")

                )



                .sessionManagement(session -> session

                        .sessionCreationPolicy(SessionCreationPolicy.ALWAYS)

                        .maximumSessions(1))



                // using customized login html page

                .formLogin((form) -> form

                        .loginPage("/login")
```

```
                    .defaultSuccessUrl("/authenticated")

                    .failureUrl("/login?error=true")

                    .failureHandler(authenticationFailureHandler())

                    .permitAll()

            )



            .logout((logout) -> logout

                    .logoutSuccessUrl("/welcome")

                    .deleteCookies("JSESSIONID")

                    .invalidateHttpSession(true)

                    .permitAll()

            );



        return http.build();

    }



    @Bean

    public AuthenticationFailureHandler authenticationFailureHandler() {

        return new CustomAuthenticationFailureHandler();
```

```
}



@Bean
public HttpSessionEventPublisher httpSessionEventPublisher() {
    return new HttpSessionEventPublisher();
}



@Bean
public UserDetailsService userDetailsService(){

    UserDetails user = User.builder()
            .username("user")
            .password(passwordEncoder().encode("userpassw"))
            .roles("USER")
            .build();

    UserDetails admin = User.builder()
            .username("admin")
            .password(passwordEncoder().encode("adminpassw"))
```

```
            .roles("ADMIN")

            .build();



    return new InMemoryUserDetailsManager(user, admin);

}



@Bean

public static PasswordEncoder passwordEncoder(){

    return new BCryptPasswordEncoder();

}



/*  to use Digest Authentication



    DigestAuthenticationEntryPoint entryPoint() {

    DigestAuthenticationEntryPoint result = new DigestAuthenticationEntryPoint();

    result.setRealmName("My Security App Realm");

    result.setKey("3028472b-da34-4501-bfd8-a355c42bdf92");

}
```

```
@Autowired
UserDetailsService userDetailsService;



DigestAuthenticationFilter digestAuthenticationFilter() {
    DigestAuthenticationFilter result = new DigestAuthenticationFilter();
    result.setUserDetailsService(userDetailsService);
    result.setAuthenticationEntryPoint(entryPoint());
}



@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
            // ...
            .exceptionHandling(e -> e.authenticationEntryPoint(authenticationEntryPoint()))
            .addFilterBefore(digestFilter());
    return http.build();
} */
```

```
}
```

Listing 5-11

SecurityConfiguration.java

Summary

This chapter covered one of the biggest concerns of the framework: web support in Spring Security. You saw that the main functionality comes in the form of servlet filters. This is a good thing from a standards point of view because it means you can leverage Spring Security web support in other frameworks that use the standard Java servlet model.

You should now know a lot of details about the main filters that build the framework, how they work internally, and how they fit within each other and with the rest of the framework. We explained it using practical, real-life scenarios.

The next chapter covers the second major concern of Spring Security—namely, method-level security. We show how it compares to web-level security. You can leverage a lot of your current knowledge to apply it to the method-level security layer.