

Spring Security

Lab 2

1	LAB SETUP	1
2	SETTING UP BASIC SPRING MVC JSP APP	1
3	INTRODUCING SPRING SECURITY DEFAULT LOGIN PAGE	3
4	CREATING A CUSTOM LOGIN PAGE AND HARDCODED AUTHENTICATION CREDENTIALS.....	7
5	CSRF TOKEN SUBMISSION AND COOKIE SESSION TRACKING	10
6	SPECIFYING URL PATTERNS WITH ANTMATCHERS AND USING SPRING SECURITY EXPRESSIONS.....	11
7	CUSTOMIZING LOGIN FAILURE AND LOGOUT	14
8	CUSTOMIZING ERROR MESSAGES, LOGIN PAGE AND ACCESSING USER INFO	16

1 Lab setup

Make sure you have the following items installed

- Latest version of JDK 8 / 11 (note: labs are tested with JDK 8 but should work on JDK 11 with no or minimal changes)
- Spring Tool Suite (STS) IDE
- Latest version of Maven
- A suitable text editor (Notepad ++)
- A utility to extract zip files

2 Setting up basic Spring MVC JSP app

The two main aspects of the Spring Security framework that we need to understand properly is its authentication and authorization features. We will demonstrate both of these based on a simple Spring MVC app using JSP for its templating technology.

The main folder for this lab is `Security-AA-Demo`

Start up STS. Switch to the Java perspective.

Go to File -> New -> Spring Starter Project. Complete it with the following details:

Name: `SecurityAADemo`
Group: `com.workshop.security`
Artifact: `SecurityAADemo`
Version: `0.0.1-SNAPSHOT`

Description: Demo Spring Security AA features

Package: com.workshop.security

Add the following dependencies:

Web -> Spring Web

Developer Tools -> Lombok

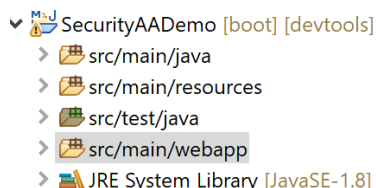
Developer Tools -> Spring Boot DevTools

Add in the following dependencies to the project POM to enable the JSP engine in Spring Boot as well as processing of JSP files. Remember to do a Maven -> Update Project.

```
<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-jasper</artifactId>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
</dependency>
```

Create a `src/main/webapp/WEB-INF` folder in the project folder structure. Make sure that `src/main/webapp` is on the project build path: Right click on the folder, select Build Path -> Use as Source Folder. This should make it appear beneath the other two folders listed beneath the main project name:



```
SecurityAADemo [boot] [devtools]
├── src/main/java
├── src/main/resources
├── src/test/java
├── src/main/webapp
└── JRE System Library [JavaSE-1.8]
```

Create a subfolder `views` within `WEB-INF` to hold the various JSP files that we will be creating next.

Replace `application.properties` with the version from changes.

This provides the prefix and suffix settings to return the correct view files from the app when it receives incoming HTTP requests.

In `src/main/resources/static`, place this file:

`favicon.ico`

This favicon will be retrieved to decorate the tab for the login page for the app that we will be introducing later.

In `src/main/webapp/WEB-INF/views`, place these files:

```
add.jsp
deleteall.jsp
deletesingle.jsp
deletesome.jsp
index.jsp
update.jsp
viewall.jsp
viewsingle.jsp
viewsome.jsp
```

In the package `com.workshop.security` in `src/main/java`, place this file:

```
WebMvcConfiguration
```

This provides a simple and direct way to map between paths and views to return corresponding to those paths. Typically, we would define this in `@GetMapping` methods in a main `@Controller` class, but since we are not implementing any business logic beyond the simple mapping, this approach will suffice for now.

Start up the app in the usual manner from the Boot dashboard.

Open a browser tab at: <http://localhost:8080/>

Click on the various links at the home page of the app and verify that they navigate to an appropriately titled page.

At this point, we have the skeleton scaffolding for a simple web app to perform management of some records without any UI or logic implemented in the individual views yet.

An important point to note here is that a production grade app will almost always require securing using HTTPS, as we demonstrate in another lab. We will skip this additional step so that we can focus solely on demonstrating the features of the Spring Security framework.

3 Introducing Spring Security default login page

We can introduce the Spring starter dependency. Right click on the project name, select Spring -> Add Starters and select Security -> Spring Security and click next. In the next window, just select `pom.xml` (DON'T choose `application.properties` as well) and click Finish

If you check the project POM now, there are two additional new dependencies that introduce the Spring Security framework into our runtime classpath:

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

Now, when we restart the app, any attempt to access any page on this app will be intercepted with the default Spring auto-generated login page at: <http://localhost:8080/login> that looks similar to the following:



The image shows a simple login form titled "Please sign in". It has two input fields: "Username" and "Password". Below the "Password" field is a "Sign in" button. The form is styled with a light gray background and rounded corners.

The default user name for this login prompt is: `user`

The password is shown in the log output from the MVC app:

Using generated security password: `xxxxxx`

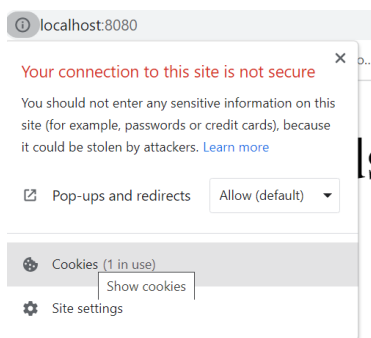
After typing in this username / password combination, you should be able to view the app's home page and navigate to the various pages as usual without any further prompting.

Restart the app. Notice a different random password is generated this time.

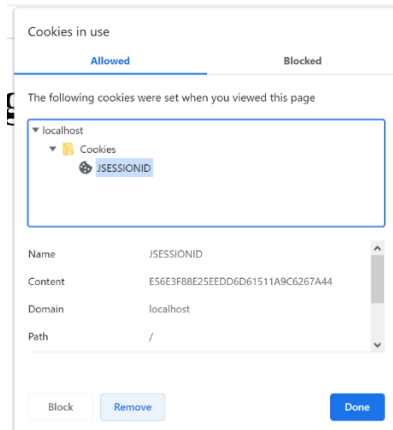
Now in a new browser tab, attempt to navigate to: <http://localhost:8080/>

Notice you are now able to access the app without having to go through the login process. This is because the browser is submitting the cookie that contains the session token it obtained from the previous successful authentication (login) on every subsequent HTTP GET request.

We will remove the cookie. Click on the address bar icon on the left, and select Show Cookies



In the cookies window, drill down to the JSESSIONID, select it, click Remove and click Done.



The JSESSIONID cookie is added in the initial response from the app to the initial GET request from the browser. The cookie contains a session token that temporarily identifies the currently authenticated user to the app and is submitted by the browser with all requests to the app. This obviates the need for the client to reauthenticate itself via the login page for every single request.

Now when you refresh the page, you should see the login page reloaded, and you will have to re-enter the latest username / password combination to authenticate yourself again. Again notice that a new password is generated for each restart.

Try entering an incorrect password sequence. You will be redirected back to the login page with an appropriate error message displayed. Notice the URL includes an additional `error` query parameter that is valueless: <http://localhost:8080/login?error>

Please sign in

Bad credentials

Username

Password

Sign in

You may notice a significant delay in the loading of the default login page. This is very likely due to a failure to download the required CSS libraries used to style it (possibly due to a server outage). If you view the Network tab while refreshing this address: <http://localhost:8080/login>, you should be able to see the affected CSS library

Name	Status	Type	Initiator	Size	Time	Waterfall
login	200	docu...	Other	1.7 kB	5 ms	
bootstrap.min.css	(pend...	styles...	login	0 B	Pendi...	
signin.css	200	styles...	login	(disk ...	1 ms	
extn-utils.html	200	docu...	jquery-3.1.1...	1.1 kB	2 ms	
extn-utils.js	200	script	chrome-ext...	1.2 kB	2 ms	

Name	Status	Type	Initiator	Size	Time	Waterfall
login	200	docu...	Other	1.7 kB	5 ms	
bootstrap.min.css	(failed)	styles...	login	0 B	21.05 s	
signin.css	200	styles...	login	(disk ...	1 ms	
extn-utils.html	200	docu...	jquery-3.1.1...	1.1 kB	2 ms	
extn-utils.js	200	script	chrome-ext...	1.2 kB	2 ms	
favicon.ico	302	/ Redi...	Other	342 B	3 ms	
login	200	text/h...	favicon.ico	0 B	2 ms	

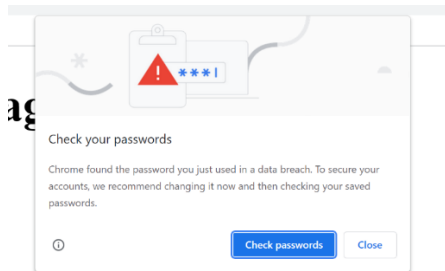
Since we cannot change this default login page that is autogenerated by the Spring Security framework, we will swap in our custom login page soon in order to speed up the process of demonstrating its security features.

Right now, let's introduce two additional properties to our `application.properties` that allow us to customize the username and password combination used in the auto-generated login page:

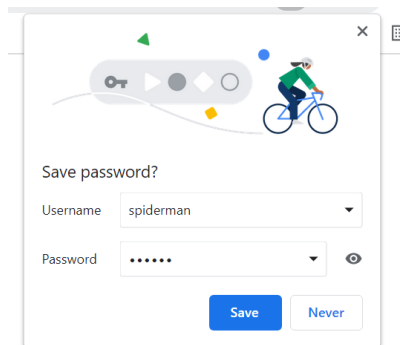
```
spring.mvc.view.prefix=/WEB-INF/views/  
spring.mvc.view.suffix=.jsp  
  
spring.security.user.name=spiderman  
spring.security.user.password=spider
```

Restart the app, and verify that you can now login with the newly specified username / password combination.

Depending on the browser you are using, a variety of messages may appear at this point. If you are using Chrome, a safety message will appear warning you about the insecurity of the password that you just entered: you can safely ignore this for now and click Close.



If you are prompted to save the password, select Never. For a real app, you could choose to save the password to avoid the hassle of reauthenticating yourself again when you access it in the future, but for this practice lab, we will skip this option.



You can experiment with a few different username and password combinations and verify that they work, remembering as well as to clear out the cookies for each connection session if you want to be able to view the default login page.

4 Creating a custom login page and hardcoded authentication credentials

We can customize most of the existing defaults that Spring Security uses for its form based authentication process. One of the primary reasons for customizing or configuring the login process is to obscure the fact that the security is implemented via Spring Security (which a hacker can figure out if all the default framework values are used). In that case, it makes it easier for a hacker to compromise the system should a zero day vulnerability be discovered in the Spring Security framework before developers have adequate time to rectify the issue.

In the package `com.workshop.security` in `src/main/java`, make this change:

`WebMvcConfiguration-v2`

Here, we simply add in an additional mapping for the `/login` path to our new `customLogin.jsp` file

In the package `com.workshop.security` in `src/main/java`, place this file:

`SecurityConfig`

In `src/main/webapp/WEB-INF/views`, place the file:

`customLogin.jsp`

Finally remove the properties for defining default username and passwords from `application.properties`.

```
spring.mvc.view.prefix=/WEB-INF/views/  
spring.mvc.view.suffix=.jsp
```

`SecurityConfig` is the primary security configuration file (which we mark with `@EnableWebSecurity`) to specify a list of username / password combinations as well as relevant methods to control authentication, authorization, secure password hashing (encoding) and a variety of other related security functionality. When we do this, we no longer need the default username and password properties defined in `application.properties` earlier.

The `configure` method with the `AuthenticationManagerBuilder` parameter is used to configure the Authentication Provider with a username / password list to be used in authenticating the login from the client end.

```
@Override  
public void configure(AuthenticationManagerBuilder auth) throws Exception {
```

Here, we are using the in-memory authentication approach (based on `InMemoryUserDetailsManager`), where we maintain the username / password list only in memory. This is suitable for rapid prototyping, but is not realistic for a production app that will need to persist and subsequently retrieve username / passwords in a secure manner, which we will examine further in a later lab.

IMPORTANT: In Spring Boot 2 and Spring 5, secure password hashing is compulsory: therefore a factory @Bean method must be specified to return a PasswordEncoder instance. The password hashing algorithm of choice must be specified via this factory method.

The various reliable password hashing options are based on KDF such as Argon2, Bcrypt, Scrypt.

<https://javadoc.io/doc/org.springframework.security/spring-security-crypto/latest/index.html>

You can return any one of their concrete implementations, for e.g.

- `org.springframework.security.crypto.argon2.Argon2PasswordEncoder`
- `org.springframework.security.crypto.scrypt.SCryptPasswordEncoder`
- `org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder`

Other options include PBKDF2 (and some older unreliable options)

[https://javadoc.io/static/org.springframework.security/spring-security-crypto/5.5.2/org.springframework.security/crypto/password/package-summary.html](https://javadoc.io/static/org.springframework.security/spring-security-crypto/5.5.2/org.springframework.security.crypto.password/package-summary.html)

Note that many of the older unreliable options are deprecated

Due to the Spring Boot 2 password encoding compulsory requirement, we must always add `passwordEncoder(passwordEncoder())` to the builder chain (at any position is fine - end or beginning of the chain), regardless of which inbuilt-mechanism we are using.

Note that we can specify both roles and authorities for a particular user. An authority (in its simplest form) is just a string, it can be anything like: `user`, `ADMIN`, `ROLE_ADMIN`. A role is an authority with a `ROLE_` prefix. So, a role called `ADMIN` is equivalent to an authority called `ROLE_ADMIN`. Keep this in mind when you are implementing code in the `configure` method to authorize requests based on roles, authorities and authenticated status, etc

The `configure` method with the `HTTPSecurity` parameter is used to configure authorization for incoming requests by pattern matching on specific URLs as well as specific HTTP methods. In addition, we can also specify further customizations of the login and logout phases of the entire authentication process.

In Spring Security, authorization can be done at the level of:

- URL - each URL maps to a view or resources, and we can specify which authenticated users are permitted to access which specific URLs using which HTTP methods (i.e. GET, POST, PUT, etc). This is the most common approach.
- Specific parts within a view - A view is typically a JSP or HTML file (e.g. a Thymeleaf template) and we can restrict access to certain parts within that view to specific authenticated users
- Methods - We can specify fine grained authorization semantics at the method level to restrict execution of certain methods to specific authenticated users

The initial part of the builder pattern:

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


specifies that all incoming requests require authentication, which means that any attempt to access any view through any valid URL path will bring up the configured login page if authentication has not yet been successful at that point in time.

The second part of the builder pattern, which is separate from the first with an `and()`

```
.formLogin()  
    .loginPage("/login").permitAll()  
    .loginProcessingUrl("/performLogin");
```

`formLogin()` specifies that a customized login form will be provided and will be made available at the path given in `loginPage` which in this case is `/login`. We could have used another path if we wanted to, but `/login` is a typical choice. We configured this path to map to `customLogin.jsp` in `WebMvcConfiguration`.

`permitAll` simply states that all users (regardless of whether they are authenticated or not) are allowed to access this view. This is absolutely necessary since everyone is unauthenticated at the point when they access the login page.

Finally `loginProcessingUrl` specifies a custom URL which will kickstart the process of Spring Security performing validation of the submitted password from the login form automatically in the background. This involves performing the specified hash algorithm (as specified by the `PasswordEncoder` call in the configuration of the Authentication Manager) on the password. Then the newly computed hash is compared against the stored hash for that matching username provided by the configured authentication mechanism.

The login form functionality is provided by `customLogin.jsp`. Some of the key requirements for this form are:

- a) The URL we submit the form to is the one specified in `loginProcessingUrl`, which in this case is `/performLogin`. As mentioned earlier, this will kickstart the process of Spring Security performing validation of the submitted password from the login form. If we omitted `loginProcessingUrl` from our custom configuration, then the URL to use instead will be the one specified in `loginPage`, which in this case would be `/login`
- b) The URL is submitted using a POST method specified inside a Spring form tag (`<form:form>`), and not a normal HTML form tag. This is necessary for creating a CSRF token for protecting against CSRF attacks (to be discussed later).
- c) The username should be present on the HTTP parameter `username` and the password should be present on the HTTP parameter `password`. Both of these are default values that will be used by Spring Security in extracting the username and password values from the submitted form.

Here, the CSS for property styling is omitted or kept to a minimum so that emphasis can be given solely to the important HTML elements necessary in the creation of the custom login form. There are many sites that provide more comprehensive styling for a realistic login form for a production app:

https://www.w3schools.com/howto/howto_css_login_form.asp

<https://www.c-sharpcorner.com/article/creating-a-simple-login-page-using-html-and-css/>

Restart the app.

Verify that:

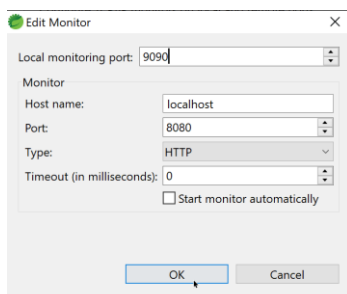
- you are presented with the custom login page when you initially access any of the valid URLs for this app.
- you can login in with any of the hardcoded credentials provided as String constants in SecurityConfig
- once logged in, you are able to access any of the valid URLs for the app
- when you restart the app after successful login, you will still be able to access any of the valid URLs. You will need to clear the cookie associated with the session in order to be presented with the custom login page again
- Login failures result in the login page being presented again with the additional query parameter error added at the end: <http://localhost:8080/login?error>

5 CSRF token submission and cookie session tracking

Locate a free port available for use on your machine. To check whether an existing process is listening on any random port, run the following command in a command prompt in Windows:

```
netstat -aon | findstr port-number
```

Assuming that port 9090 is free, set up Eclipse TCP/IP monitor to monitor this port and redirect traffic to localhost:8080 (where the app is currently running at)



Clear the cookie from the browser tab, restart the app and navigate to:

<http://localhost:9090/login>

Check the log in the monitor. Notice that

- The response to the initial request returns a JSESSIONID cookie of a specific value via the Set-Cookie header and the response body includes the HTML of the login form as well as a CSRF value provided as `_csrf` parameter in a hidden input field. Both the cookie and CSRF value are randomly generated by Spring Security on behalf of the app.
- The subsequent automatic request to the app for `favicon.ico` now includes the JSESSIONID cookie provided in the initial response.

Type in a valid username / password combination in the login form and click Log In. Notice that:

- The request to `/performLogin` is a POST which includes the JSESSIONID cookie again as well as the CSRF token in the body of the request alongside the username and password values. Without the CSRF token, Spring Security will not process the login form (to protect against possible CSRF attacks).
- The response includes a new JSESSIONID cookie with a new value that indicates a new active session for the current authenticated user.

Type in any valid URL in the address bar to the monitor at port 9090, for e.g:

<http://localhost:9090/view/single>

Notice again that the outgoing request includes the new JSESSIONID cookie value to identify the authenticated user for the current active session.

Restart the app. Type in the same URL again:

<http://localhost:9090/view/single>

Notice again that the outgoing request includes the same JSESSIONID cookie value as the previous request. Even with the app restarted, it still recognizes the cookie from the previous session. This is not properly secure, and we need to implement a logout to invalidate the cookie from being recognized and processed by the Spring Security framework.

Delete the cookie from the browser tab. Type in the same URL again:

<http://localhost:9090/view/single>

Notice now that since there is no longer any JSESSIONID cookie included in the outgoing request, the app assumes that this is new session and again returns the custom login form to authenticate the user.

As an aside, we are able to sniff HTTP traffic between the browser and the app using the TCP/IP monitor because the connection is unencrypted. In a production app, we would definitely want to ensure a HTTPS connection is established to protect the confidentiality of passwords that are transmitted in plaintext form to the server.

6 Specifying URL patterns with antMatchers and using Spring Security expressions

At this point, we have simply allowed any authenticated use to access any resource (URL) in the app. We are now going to fine tune our access control to only allow access to specific URLs for specific users.

In the package `com.workshop.security` in `src/main/java`, make the following change:

`SecurityConfig-v2`

The general structure for URL authorization in the `configure` method is as follows:

```
http.authorizeRequests()
    .antMatchers(URL-pattern-1).security-expression-1
    .antMatchers(URL-pattern-2).security-expression-2
...
...
    .antMatchers(URL-pattern-n).security-expression-n
    .and()

... configuration for login form customization ...
```

Examples of possible URL patterns for Ant pattern matching:

<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/util/AntPathMatcher.html>

Examples of typical security expressions:

hasRole, hasAnyRole, hasAuthority, hasAnyAuthority, permitAll, denyAll, anonymous, authenticated

Let's examine all the authorization rules in the builder chain one at a time:

```
.antMatchers("/view/all").permitAll() // rule 1
.antMatchers("/delete/all").denyAll() // rule 2
.antMatchers("/view/**").hasAnyRole(role1, role2) // rule 3
.antMatchers("/add", "/update").hasRole(role1) // rule 4
.antMatchers("/delete/**").hasRole(role2) // rule 5
.anyRequest().authenticated() // rule 6
```

- 1) All requests to the URL `/view/all` will be permitted without any authentication required
- 2) All requests to the URL `/delete/all` will be denied, regardless of who the user is (even if the user is authenticated)
- 3) All requests to any URLs nested within `/view` will be allowed to any authenticated users that have either the role of `role1` or `role2`
- 4) All requests to the URLs `/add` and `/update` are only allowed to any authenticated users with the role of `role1`
- 5) All requests to any URLs nested within `/delete` are only allowed to any authenticated users with the role of `role2`
- 6) All requests to any URLs must be authenticated

IMPORTANT: The order of the `antMatchers()` elements is significant; the first matching rule will override other rules later in the builder chain. Therefore, typically more specific rules need to come first, followed by the more general ones (although this is not the case in this example).

For e.g.

- rule 1 overrides rule 3 and rule 6
- rule 2 override rule 5

Clear the browser cookie and attempt to access the following URLs in turn and observe the results. You can use Eclipse TCP/IP monitor if you wish to view the CSRF tokens and session cookie values: in that case, use `localhost:9090` instead.

<http://localhost:8080/view/all>

The view is returned without any need to authenticate (Rule 1)

<http://localhost:8080/view/some>

The login form is returned due to Rule 3 which requires authenticating the user first in order to determine their role. Login with any user that is in **role1**. The requested page is returned.

<http://localhost:8080/add>

The requested page is returned without the need for any further login, as the currently authenticated user is in **role1** and the session cookie is still active.

<http://localhost:8080/delete/some>

At this point a Whitelabel error page with status code 403 Forbidden is returned. This is because the currently authenticated user is not in the correct role (**role2**) to access this view (Rule 5). Notice that the option to login again is not presented. So we will have to manually do this in the next request.

<http://localhost:8080/login>

Login now with any user that is in **role2**

Notice that you are now presented with the home page. Whenever a successful authentication occurs at the custom login page without a previous request to some other page first (i.e. the request was made directly to the login page as is the case here), Spring Security will by default redirect you to the root path of the app (\), which in this case is mapped to `index.jsp` in `WebMvcConfiguration`

Click on Delete Some Records. This time you should be able to retrieve the page successfully.

Return to the home page and click on Delete All Records.

Again, you will be presented with a Whitelabel error page with status code 403 Forbidden, as Rule 2 denies access to everyone for this specific URL.

Click back to return to the home page and click on any of the links in the View Records section. You should be able to retrieve all of them (Rule 3).

Next click on Add a Single Record. Again, you will be presented with a Whitelabel error page with status code 403 Forbidden, as Rule 4 limits access to this URL to users who are in **role1**

<http://localhost:8080/login>

Login now with any user that is in **role1**

At the home page, click on Add a Single Record. This time you should be able to retrieve the page successfully (Rule 4). Repeat this for Update a Single Record

Play around with navigating through the various links and login form to check how the authorization rules are applied.

The term roles and authorities that are used to categorize a user are closely related. An authority is just a string, it can be anything like: `ADMIN`, `ROLE_ADMIN`. A role is an authority with a `ROLE_` prefix. So, a role called `ADMIN` is equivalent to an authority called `ROLE_ADMIN`.

In the package `com.workshop.security` in `src/main/java`, make the following change:

`SecurityConfig-v3`

This implementation is functionally equivalent to the previous version based on the explanation on the relationship between roles and authorities:

```
.antMatchers("/view/all").permitAll() // rule 1
.antMatchers("/delete/all").denyAll() // rule 2
.antMatchers("/view/**").hasAnyAuthority(rolePrefix + role1,
rolePrefix + role2) // rule 3
.antMatchers("/add", "/update").hasAuthority(rolePrefix + role1)
// rule 4
.antMatchers("/delete/**").hasAuthority(rolePrefix + role2) //
rule 5
.anyRequest().authenticated() // rule 6
```

You can go through the previous testing sequence to verify this for yourself.

7 Customizing login failure and logout

We continue by further customizing the handling of a login failure and also implement a logout to properly invalidate the HTTP session.

Before doing that, verify the following features that we are now going to change with the next set of modifications:

- Login failures result in the login page being presented again with the additional query parameter error added at the end: <http://localhost:8080/login?error>
- When the app is restarted, the previous session is still active in that the previously authenticated user is still able to access whatever URLs that is authorized for her via the session cookie submitted from the browser.

In the package `com.workshop.security` in `src/main/java`, make this change:

`SecurityConfig-v4`

In `src/main/webapp/WEB-INF/views`, make this change:

`index-v2.jsp`

In `SecurityConfig`, we have added an additional rule (Rule 0) to allow all users to access the root path. We have also make further extensions to the builder chain for customizing login and logout.

```
.formLogin()
.loginPage("/login").permitAll()
.loginProcessingUrl("/performLogin")
.failureUrl("/login?error=true")

.and()
```

```
.logout()  
    .logoutUrl("/performLogout")  
    .logoutSuccessUrl("/login?logout=true").permitAll();
```

- `failureUrl` - specifying the URL to redirect to in the event of a login failure
- `logout` - specifying that logout customization will be implemented
- `logoutUrl` - specifying the URL to send a POST request to in order to trigger Spring Security to perform a logout action by invalidating the current HTTP session
- `logoutSuccessUrl` - specifying the URL to redirect to after the logout has completed. Typically, the convention is to redirect to the login page so that the user has the opportunity to log in again immediately

In `index.jsp`, we add an additional POST method specified inside a Spring form tag (`<form:form>`), and not a normal HTML form tag, to the path specified in the `logoutUrl`

```
<form:form method="POST" action="/performLogout">  
    <button type="submit">Log out</button>  
</form:form>
```

Restart the app.

Remove the cookie in the browser and navigate to:

<http://localhost:8080>

Click on Add a Single Record. You will be redirected to the login page.

Enter an invalid credential combination. Notice now you are redirected to:

<http://localhost:8080/login?error=true>

Enter a valid credential for any user with `role1`

You should now be able to retrieve the page at: <http://localhost:8080/add>

Return to the main menu and click Logout.

You will be redirected to the login page at the URL:

<http://localhost:8080/login?logout=true>

In the same browser tab, attempt to navigate back to: <http://localhost:8080/add>

You are no longer able to do so as the logout has invalidated the session, even though you have not deleted the cookie from the previous session and this cookie is submitted to the app in with every browser request.

You can go through the testing sequence of the previous section to check that all the other authorization rules are still operating as normal.

8 Customizing error messages, login page and accessing user info

At the moment, the login page appears in 3 different situations with 3 different URLs:

Normal login: <http://localhost:8080/login>

Login failure: <http://localhost:8080/login?error=true>

Logout complete: <http://localhost:8080/login?logout=true>

In all situations, the login page appears identical. We can customize it based on the appearance of the query parameters in the URL to make it easier for the user to interact with it.

We may also wish to return customized error pages instead of the Whitelabel error pages.

Lastly, it would be useful to be able to access info pertaining to the currently authenticated user for any necessary business logic that we may need to perform in our app. For this, we will create a proper `@Controller` class and shift some of the URL mappings out from the `WebMvcConfiguration` to individual `@GetMapping` methods in there.

In the package `com.workshop.security` in `src/main/java`, make this change:

```
WebMvcConfiguration-v3
```

and place these new files:

```
EmployeeController  
CustomErrorController
```

In `src/main/webapp/WEB-INF/views`, make these changes:

```
customLogin-v2.jsp  
add-v2.jsp  
deletesome-v2.jsp
```

In `src/main/webapp/WEB-INF/views`, create the folder `error` and place the following files in it:

```
403.jsp  
404.jsp
```

We remove the mappings for certain paths from `WebMvcConfiguration` and implement them in `EmployeeController` instead. The `addPage` and `deleteSomePage` methods demonstrate how to extract the username and authority of the currently authenticated user, and then embed them in an attribute to be accessed by the JSP view that they return respectively (`add.jsp` and `deletesome.jsp`)

The `loginPage` returns the custom login page with an appropriate message depending on the query parameter present in the path (either `error` for login failure or `logout` for successful logout). This message is displayed in the returned page (`customLogin.jsp`)

The `CustomErrorController` implements the `ErrorController` marker interface with a mapping for the `/error` path, which automatically takes precedence over the default Whitelabel error page. We then determine the exact status code of the error and then return an appropriate view (placed in the

error folder in `src/main/webapp/WEB-INF/views`). These views are `403.jsp` and `404.jsp`. Note that we could also have done a similar view implementation for `500.jsp`, but we did not.

Restart the app.

Proceed to <http://localhost:8080/login> to login

Experiment with entering invalid credentials, then login with valid credentials. Return to home page and log out. Verify that the messages that appear on the custom login page corresponds correctly to the specific situations.

Login with a user that has *role1*. Attempt to access the Add a Single Record page (<http://localhost:8080/add>) and verify that the user name is shown correctly.

Login with a user that has *role2*. Attempt to access the Delete some records page (<http://localhost:8080/delete/some>) and verify that the user name and authority is shown correctly.

Attempt to access a page that the current authenticated user is not authorized to access. Verify that the appropriate custom error page for code 403 is returned.

Attempt to access an invalid URL by simply typing it in the address bar (for e.g. <http://localhost:8080/asdfasdf>) . Verify that the appropriate custom error page for code 404 is returned.