Spring Security (SEC)

Exercise SEC.1 Spring Security

The Setup:

Create a copy of **W3D1-MVC**, and call it **W3D2-SEC**. Remember to also change the project name inside the pom.xml's `<artifactId>` tag.

Inside webapp/META-INF/context.xml you may also want the change "/mvc1" to "**/sec**" to that when you deploy it on Tomcat it also uses a different url.

Add the dependencies outlined in the W3D1-MVC exercise, and then also add the following additional dependencies for spring security:

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>5.6.1</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-taglibs</artifactId>
    <version>5.6.1</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>5.6.1</version>
</dependency>
```

The Application:

The application is the same Car (or Book if you copied your soluation) application as used in the MVC exercise, running it should still give the same output as before.

For reference you can also look at the security code inside W1D1-COV1 and W1D1-COV2.

The Exercise:

Let's add security to our application! We will require USER authorization to see the list of cars, and ADMIN authorization in order to add, delete, or update the list.

Part A: the most important steps are:

1. Creating a SecurityConfig.java class that extends WebSecurityConfigurerAdapter (leave it empty for now), but add the @Configuration and @EnableWebSecurity annotations.
2. Inside MyWebAppInitializer add SecurityConfig.class to the rootContext.register() method as an extra parameter
3. Also inside MyWebAppInitializer add the SpringSecurityFilterChain, mapping it to "/*" (also shown on the slides)
4. Then inside SecurityConfig override the configure(AthenticationManagerBuilder auth) method to create an inMemoryAuthentication provider with plain text usernames and passwords. Be sure to create a login that only has a USER role and a login that has both a USER and a ADMIN role.
5. Next override the configure(HttpSecurity http) method, using .antMatchers() to indicate which urls can be accessed anonymously (if any), which urls require a USER role, and which urls require an ADMIN role.

6. Remember to also add the .formLogin().and().logout(). Perhaps even put .permitAll() after the .formLogin() to make sure your users can get to the form.
7. The bookDetail.jsp page currently doesn't have a CSRF token inside the form. Use the Spring Security tags to add it.
8. Test your application to see if it works (cannot see resources if not logged in, USER can only see the list, ADMIN can do everything else).
9. In case you get 403 Forbidden errors, then be sure to override the 3rd method inside securityConfig to add the debug option, and to create a log4j2.xml file inside the resources directory with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
        </Console>
    </Appenders>
    <Loggers>
        <Root level="warn">
            <AppenderRef ref="Console"/>
        </Root>
        <Logger name="org.hibernate" level="warn">
            <AppenderRef ref="Console"/>
        </Logger>
        <Logger name="org.springframework.security" level="debug">
            <AppenderRef ref="Console"/>
        </Logger>
    </Loggers>
</Configuration>
```

Part B:

10. Lets update bookList.jsp so that we can logout. We can do so by adding the following link. The easiest place is probably just underneath the add Car link (line 26)

```
<a href="logout">Logout</a>
```

11. Next we're going to hide the add and edit link for people with the USER role (after all they're not allowed to go there). Start by adding the security taglib to the top of the page.

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
```

12. Then add the <sec:authorize> tag shown below around the edit and the add links so that they are only shown if the user hasRole('ADMIN')

```
<sec:authorize access="hasRole('ADMIN')">
```

13. Test your application and log in as a user to see if the links are gone, then logout and log back in as an admin and check if you can correctly use the pages.

Part C (if time allows – be sure do validation first):

- Switch to using a JDBC Authenticator instead of plain text
- Try with both the standard tables and the non-standard tables
- As a last step create your own custom login form. If you want you can copy and paste the custom login form I had on the slides, or make your own. Just be sure that the configuration values match the values used for username and password on the page.

Validation (VAL)

Exercise VAL.1

The Setup

Create a copy of your solution to W3D2-SEC (or if you haven't finished that yet, make a copy of W3D1-MVC solution) and call it **W3D2-VAL**.

Remember to change the project name inside the pom.xml's `<artifactId>` tag, and update the deployment path Inside webapp/META-INF/context.xml to "**/val**"

Add the following dependency for the hibernate validator:

```
</dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.2.1.Final</version>
</dependency>
```

You can look inside W1D1-COV1 and W1D1-COV2 as examples of projects using validation.

The Exercise

Part A: add validation to the book class.

- Inside the Book class add the @ISBN annotation to the ISBN property, and add @Positive to the price property. Then add @NotBlank to the title and author fields.
- Update bookDetail.jsp to use the Spring form tags to display there error messages. Remember to add taglib at the top, and then update the form tags. The <form:form> tag also automatically includes the CSRF token, which means you can remove the <sec:csrfInput /> tag.
- Update the book controller by adding the @Valid annotation before the Book parameter in the add() and update() methods, and also add a BindingResult parameter right after it.
- The quickest way to test the validation is to not do POST/Redirect/GET. Update the code inside the add() and update() methods to check if your bindingResult .hasErrors() and if it does simply return "bookDetail", which will automatically display the errors.
- Test your application to see if everything works. Hint: @ISBN can be very strict, I recommend using the following ISBN number for testing: 978-0-306-40615-7

Part B: POST/Redirect/GET:

- Inside both the add() and update() the method where you check if the BindingResult has errors, make it use RedirectAttributes to store the BindingResult inside a FlashAttribute. In a second FlashAttribute also store the book object. **Important**: if you store your book inside a flashAttribute with the key "book", then you need to store the BindingResult with key: `org.springframework.validation.BindingResult.book`
- The add method should then redirect back to "redirect:/books/add" and the update method to "redirect:/books/{id}".
- On the methods that implement the @GetMapping for these object we have to make sure that we do not overwrite the book object that the FlashAttribute brings in (as it will also stop our bindingResult / errors from showing).
- What this means is the viewAdd() method has to remove @ModelAttribute Book book from its parameters, and instead add the following code to its contents:

```
if (!model.containsAttribute("book")) {
  model.addAttribute("book", new Book());
}
```

- Similarly the get() method should surround its first line (where it adds the book to model) with the same if statement (that checks if book is not yet inside mode).
- Test your application again to see if everything works.

Optional Additional Exercise:

- Add a published date to the book class, and validate that it’s in the @Past