Spring MVC Security
User Registration Tutorial

Introduction

In this tutorial, you will learn how to perform user registration with Spring Security.

We'll create a user registration form and store the user's information in the database. We'll also cover the steps of encrypting the user's password using Java code.

Prerequisites

This tutorial assumes that you have already completed the Spring Security videos in the Spring Boot, Spring and JPA/Hibernate course. This includes the Spring Security videos for JDBC authentication for plain-text passwords and encrypted passwords.

Overview of Steps

1. Download and Import the code
2. Run database scripts
3. Validation support to Maven POM
4. WebUser class
5. Security Configs
6. Button on login page for "Register New User"
7. Registration Form
8. Registration Controller
9. Confirmation Page
10. Test the App
11. Verify User Account in the Database

1. Download and Import the Code

The code is provided as a full Maven project and you can easily import it into Eclipse.

DOWNLOAD THE CODE

1. Download the code using link from the lecture.
2. Unzip the file

IMPORT THE CODE

1. In IntelliJ, select **File > Open**
2. Browse to directory where you unzipped the code.
3. Click OK buttons etc to import code

REVIEW THE PROJECT STRUCTURE

Make note of the following directories

- /src/main/java: contains the main java code
- /src/main/resources/templates: contains the Thymeleaf files
- /sql-scripts: the database script for the app (security accounts)

2. Run database scripts

In order to make sure we are all on the same page in terms of database schema names and user accounts/passwords, let's run the same database scripts.

MySQL workbench

In MySQL workbench, run the following database script:

```
/sql-scripts/
02-setup-spring-security-demo-database-hibernate-bcrypt.sql
```

This script creates the database schema: **spring_security_demo_bcrypt**. The script creates the user accounts with encrypted passwords. It also includes the user roles.

User ID	Password	Roles
john	fun123	EMPLOYEE
mary	fun123	EMPLOYEE, MANAGER
susan	fun123	EMPLOYEE, ADMIN

3. Validation support to Maven POM

In this app, we are adding validation. We want to add some basic validation rules to the registration form to make sure the user name and password are not empty.

As a result, we have an entry for the Hibernate Validator in the pom.xml file.

```xml
File: pom.xml

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

4. WebUser class

For our registration form, we are using a WebUser class for the project. It will have the user name and password. We are also adding annotations for validating the fields.

File: WebUser.java

```
package com.luv2code.springboot.demosecurity.user;

import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Size;

public class WebUser {

        @NotNull(message="is required")
        @Size(min=1, message="is required")
        private String userName;

        @NotNull(message="is required")
        @Size(min=1, message="is required")
        private String password;

        // constructor, getters/setters omitted for brevity
        …

}
```

5. Security Configs

In our security configuration file, DemoSecurityConfig.java, we create a DaoAuthenticationProvider bean. This is based on our UserService. It provides access to the database for creating users. We provide the password encoder for bcrypt.

We'll also use the UserService and UserDao to check if a user exists.

The UserDao has the low-level code for accessing the security database.

File: DemoSecurityConfig.java

```
@Bean
public DaoAuthenticationProvider authenticationProvider(UserService userService) {
    DaoAuthenticationProvider auth = new DaoAuthenticationProvider();
    auth.setUserDetailsService(userService); //set the custom user details service
    auth.setPasswordEncoder(passwordEncoder()); //set the password encoder - bcrypt
    return auth;
}
```

We'll use this bean later in the Registration Controller.

The application also uses an AuthenticationSuccessHandler. The AuthenticationSuccessHandler is a special interface that is processed by Spring Security once login/authentication is successful.

In this class, we implement the AuthenticationSuccessHandler interface and add our own custom code. We'll retrieve the user from the database and then place it in the session. This is executed for every successful user login. Hence the user info will be in the session and we'll be able to display custom fields such as firstName, lastName etc.

Here's the code for the AuthenticationSuccessHandler implementation.

File: CustomAuhenticationSucessHandler.java

```java
@Controller
public class CustomAuthenticationSuccessHandler implements AuthenticationSuccessHandler {

    private UserService userService;

    public CustomAuthenticationSuccessHandler(UserService theUserService) {
        userService = theUserService;
    }

    @Override
    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse
response, Authentication authentication)
            throws IOException, ServletException {

        System.out.println("In customAuthenticationSuccessHandler");

        String userName = authentication.getName();

        System.out.println("userName=" + userName);

        User theUser = userService.findByUserName(userName);

        // now place in the session
        HttpSession session = request.getSession();
        session.setAttribute("user", theUser);

        // forward to home page
        response.sendRedirect(request.getContextPath() + "/");
    }

}
```

This code will perform the following steps:

- Retrieve the user from the database via the UserService

- Place the user in the session

- Forward to the home page

We also need to configure Spring Security to use this new class.

File: DemoSecurityConfig.java (snippet)

```java
@Bean
public SecurityFilterChain filterChain(HttpSecurity http, AuthenticationSuccessHandler
customAuthenticationSuccessHandler) throws Exception {

    http.authorizeHttpRequests(configurer ->
                    configurer
                            .requestMatchers("/").hasRole("EMPLOYEE")
                            .requestMatchers("/leaders/**").hasRole("MANAGER")
                            .requestMatchers("/systems/**").hasRole("ADMIN")
                            .requestMatchers("/register/**").permitAll()
                            .anyRequest().authenticated()
            )
            .formLogin(form ->
                    form
                            .loginPage("/showMyLoginPage")
                            .loginProcessingUrl("/authenticateTheUser")
                            .successHandler(customAuthenticationSuccessHandler)
                            .permitAll()
            )
            .logout(logout -> logout.permitAll()
            )
            .exceptionHandling(configurer ->
                    configurer.accessDeniedPage("/access-denied")
            );

    return http.build();
}
```
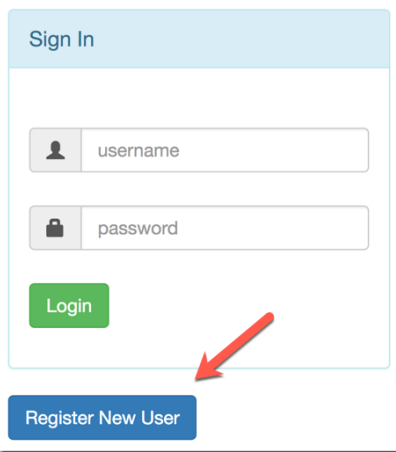
Note the reference of code:

1. .successHandler(customAuthenticationSuccessHandler)

This tells Spring Security to execute the customAuthenticationSuccessHandler for each successful login.

6. Button on login page for "Register New User"

On the login form, **fancy-login.html**, we are adding a new button for **Register New User**. This will link over to the registration form.

Near the bottom of the login form, see the new code.

File: /src/main/resources/templates/fancy-login.html

```
<div>
    <a th:href="@{/register/showRegistrationForm}"
        class="btn btn-primary mt-3"
        role="button"
        aria-pressed="true">Register New User</a>
</div>
```

7. Registration Form

We have a new form for registering a user.

This form is very similar to the login form, the main difference is that we're pointing to `/register/processRegistrationForm`. We're also making use of a model attribute for `WebUser`.

Below are the relevant snippets from the form.

File: /src/main/resources/templates/register/registration-form.html

```html
<form action="#" th:action="@{/register/processRegistrationForm}"
      th:object="${webUser}"
      method="POST" class="form-horizontal">

      <!-- User name -->
      <div style="margin-bottom: 25px" class="input-group">
          <input type="text" th:field="*{userName}" placeholder="Username (*)"
      class="form-control" />
      </div>

      …
  <!-- Registration Button -->
  <div style="margin-top: 10px" class="form-group">
      <div class="col-sm-6 controls">
          <button type="submit" class="btn btn-primary">Register</button>
      </div>
  </div>

</form>
```

8. Registration Controller

The `RegistrationController` is responsible for registering a new user. It has two request mappings:

1. `/register/showRegistrationForm`
2. `/register/processRegistrationForm`

Both mappings are self-explanatory.

REGISTRATIONCONTROLLER

The coding for the controller is in the following file.

File: RegistrationController.java

```java
@Controller
@RequestMapping("/register")
public class RegistrationController {

        ...

}
```

Since this is a large file, I'll discuss it in smaller sections.

USERSERVICE

In the `RegistrationController`, we inject the `UserService` with the code below:

```java
@Autowired
public RegistrationController(UserService userService) {
        this.userService = userService;
}
```

The `UserService` provides access to the database via the `UserDao` for creating users. We'll also use this bean to check if a user exists.

INITBINDER

The `@InitBinder` is code that we've used before. It is used in the form validation process. Here we add support to trim empty strings to null.

```java
@InitBinder
public void initBinder(WebDataBinder dataBinder) {

        StringTrimmerEditor stringTrimmerEditor = new StringTrimmerEditor(true);

        dataBinder.registerCustomEditor(String.class, stringTrimmerEditor);
```

```
        }
```

SHOW REGISTRATION FORM

The next section of code is the request mapping to show the registration form. We also create a `WebUser` and add it as a model attribute.

```
        @GetMapping("/showRegistrationForm")
        public String showMyLoginPage(Model theModel) {

                theModel.addAttribute("webUser", new WebUser());

                return "register/registration-form";
        }
```

PROCESS REGISTRATION FORM

On the registration form, the user will enter their user id and password. The password will be entered as plain text. The data is then sent to the request mapping: `/register/processRegistrationForm`

```
        @PostMapping("/processRegistrationForm")
        public String processRegistrationForm(
                    @Valid @ModelAttribute("webUser") WebUser theWebUser,
                    BindingResult theBindingResult,
                    HttpSession session, Model theModel) {

            String userName = theWebUser.getUserName();
            logger.info("Processing registration form for: " + userName);

            // form validation
             if (theBindingResult.hasErrors()){
                    return "register/registration-form";
             }

            // check the database if user already exists
            User existing = userService.findByUserName(userName);
            if (existing != null){
              theModel.addAttribute("webUser", new WebUser());
                    theModel.addAttribute("registrationError", "User name already
exists.");

                logger.warning("User name already exists.");
                return "register/registration-form";
             }

             // create user account and store in the databse
            userService.save(theWebUser);

            logger.info("Successfully created user: " + userName);

            // place user in the web http session for later use
            session.setAttribute("user", theWebUser);

           return "register/registration-confirmation";
        }
```

This code performs validation, checks the database to see if the user already exists, creates user and stores in database and finally place in the session for later use.

ENCRYPT THE PASSWORD

In the UserService, we have code to encrypt the password.

```
User user = new User();

// assign user details to the user object
user.setUserName(webUser.getUserName());
user.setPassword(passwordEncoder.encode(webUser.getPassword()));
user.setFirstName(webUser.getFirstName());
user.setLastName(webUser.getLastName());
user.setEmail(webUser.getEmail());
user.setEnabled(true);
```

We make use of the BCrypt password encoder created earlier.

The variable `webUser` has the form data the user entered. The password is the plain text password from the form. We use the BCrypt password encoder to encrypt this password.

9. Confirmation Page

The confirmation page is very simple. It contains a link to the login form.

File: /src/main/resources/templates/registration/registration-confirmation.jsp

```html
<body>

    <h2>User registered successfully!</h2>

    <!-- display first name, last name and email -->
    <ul>
        <li th:text="'User name: ' + ${webUser.userName}"></li>
        <li th:text="'First name: ' + ${webUser.firstName}"></li>
        <li th:text="'Last name: ' + ${webUser.lastName}"></li>
        <li th:text="'Email name: ' + ${webUser.email}"></li>
    </ul>

    <hr>

    <a th:href="@{/showMyLoginPage}">Login with new user name</a>

</body>
```
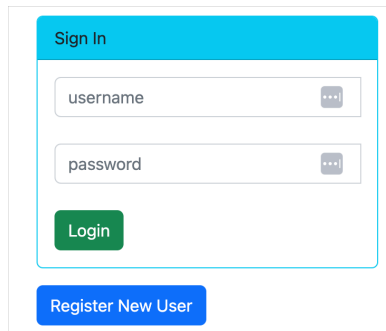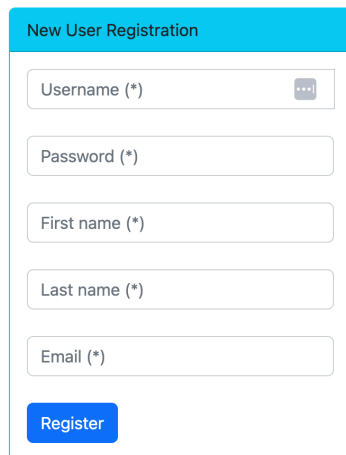
The user can now log in with the new account username. ☺

10. Test the App

At this point, you can test the application.

1. Run the app on your server. It will show the login form.

2. Click the button: **Register New User**
    a. This will show the registration form

3. In the registration form, enter a new user name and password. For example:

    - username: **tim**
    - password: **abc**

4. Click the **Register** button.

This will show the confirmation page.

5. Now, click the link **Login with new username**.

6. Enter the username and password of the user you just registered with.

7. For a successful login, you will see the home page.

luv2code Company Home Page

Welcome to the luv2code company home page!

User: Tim

Role(s): [ROLE_EMPLOYEE]

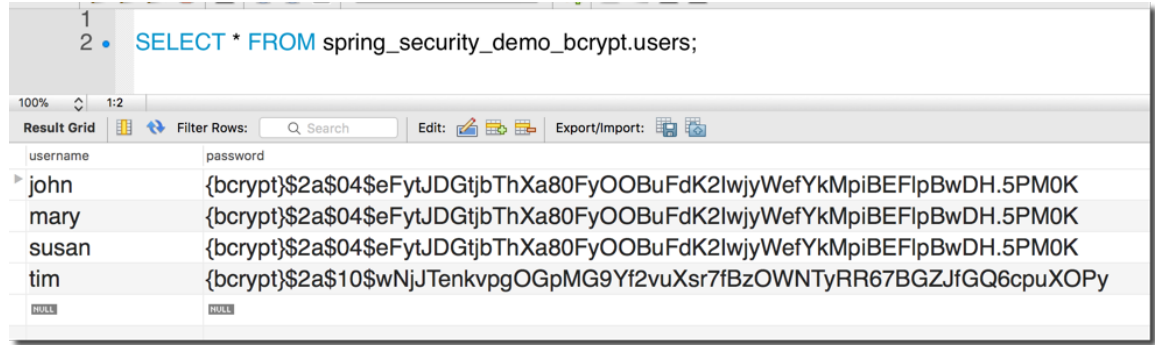First name: Tim, Last name: Smith, Email: tim@test.com

Logout

Congratulations! You were able to register a new user and then log in with them ☺

11. Verify User Account in the Database

Let's verify the user account in the database. We need to make sure the user's password is encrypted.

1. Start MySQL Workbench

2. Expand the schema for: **employee**

3. View the list of users in the **user** table.

4. You should see your new user along with their encrypted password.

```
  1
  2 •  SELECT * FROM spring_security_demo_bcrypt.users;
```

username	password
john	{bcrypt}$2a$04$eFytJDGtjbThXa80FyOOBuFdK2lwjyWefYkMpiBEFlpBwDH.5PM0K
mary	{bcrypt}$2a$04$eFytJDGtjbThXa80FyOOBuFdK2lwjyWefYkMpiBEFlpBwDH.5PM0K
susan	{bcrypt}$2a$04$eFytJDGtjbThXa80FyOOBuFdK2lwjyWefYkMpiBEFlpBwDH.5PM0K
tim	{bcrypt}$2a$10$wNjJTenkvpgOGpMG9Yf2vuXsr7fBzOWNTyRR67BGZJfGQ6cpuXOPy
NULL	NULL

Success! The user's password is encrypted in the database!