

# 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.

The diagram illustrates the user registration process. On the left, a 'Sign In' form contains fields for 'username' and 'password', a 'Login' button, and a 'Register New User' button. A red arrow points from the 'Register New User' button to a 'New User Registration' form on the right. The 'New User Registration' form includes fields for 'Username (\*)', 'Password (\*)', 'First name (\*)', 'Last name (\*)', and 'Email (\*)', along with a 'Register' button.

### 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 from:  
<http://www.luv2code.com/spring-security-user-registration>
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/setup-spring-security-bcrypt-demo-database.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.

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: /src/main/java/com/luv2code/springboot/demosecurity/user/WebUser.java

```
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.

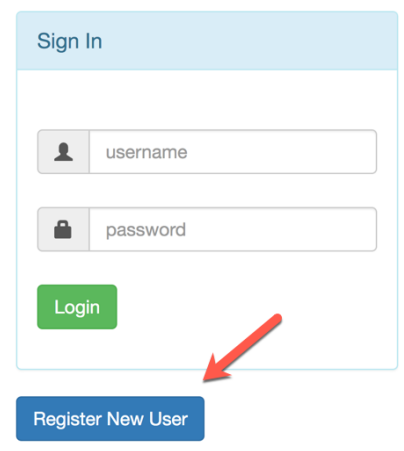
File: /src/main/java/com/luv2code/springboot/demosecurity/config/DemoSecurityConfig.java

```
@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.

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

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



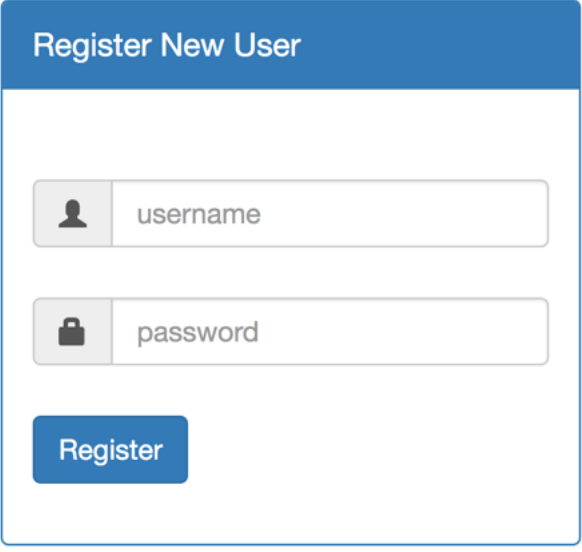
Near the bottom of the login form, see the new code.

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`

```
<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:  
`/src/main/java/com/luv2code/springboot/demosecurity/controller/RegistrationController.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:

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.

### BCRYPTPASSWORDENCODER

Next, we know that we need to encrypt passwords with `BCrypt`.

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

This code will create the `BCryptPasswordEncoder`. This class is part of the Spring Security framework. We will use it to encrypt the passwords from the user. When the

user enters their password on the registration form, we will encrypt the password first and then store it in the database. We'll see that coding shortly.

### 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.

```
@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`

The `processRegistrationForm()` method is the main focus of this bonus lecture. At a high-level, this method will do the following:

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

    // form validation

    // check the database if user already exists

    // encrypt the password

    // create user details object

    // save user in the database
    ...

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

Now let's break it down a bit and fill in the blanks.

## FORM VALIDATION

The first section of code handles form validation.

```
// form validation
if (theBindingResult.hasErrors()) {

    theModel.addAttribute("webUser", new WebUser());
    theModel.addAttribute("registrationError",
        "User name/password can not be empty.");

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

This code is in place to make sure the user doesn't enter any invalid data.

At the moment, our `WebUser.java` class has validation annotations to check for empty user name or passwords. This is an area for more improvement, we could add more robust validation rules here. But for the purpose of this bonus lecture, this is sufficient to get us going.

## CHECK IF USER ALREADY EXISTS

Next, we need to perform additional validation on user name.

```
private boolean doesUserExist(String userName) {  
  
    // check the database if the user already exists  
    boolean exists = userDetailsManager.userExists(userName);  
  
    return exists;  
}
```

This code checks the database to see if the user already exists. Of course, we don't want to add users with same user name. Granted, the database will throw back an exception if we tried this, but let's handle for this gracefully by checking first.

This method makes use of the `userService` bean that was `@Autowired` earlier in this `RegistrationController`. It has a handy method: `userExists` that will do the work for us.

Whew! We've covered the validations, now we can get down to the real business of adding the user 😊

## 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());
```

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

```
<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>
```

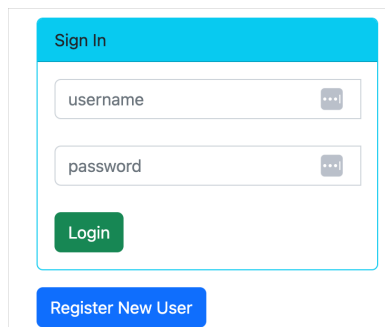
The user can now log in with the new account username. ☺

## 10. Test the App

---

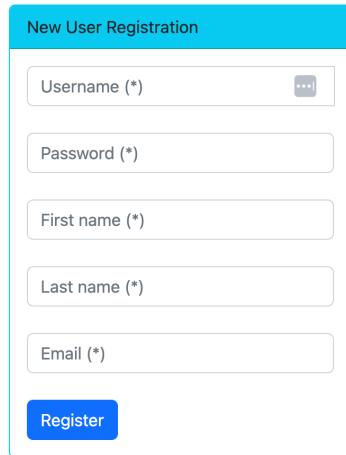
At this point, you can test the application.

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

A screenshot of a web form titled "Sign In" in a blue header. The form contains two input fields: "username" and "password", each with a small eye icon to its right. Below these fields is a green "Login" button. At the bottom of the form is a blue button labeled "Register New User".

2. Click the button: **Register New User**

- a. This will show the registration form



The image shows a 'New User Registration' form. It has a blue header with the text 'New User Registration'. Below the header are five input fields: 'Username (\*)', 'Password (\*)', 'First name (\*)', 'Last name (\*)', and 'Email (\*)'. Each field has a small icon to its right. At the bottom of the form is a blue button labeled 'Register'.

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

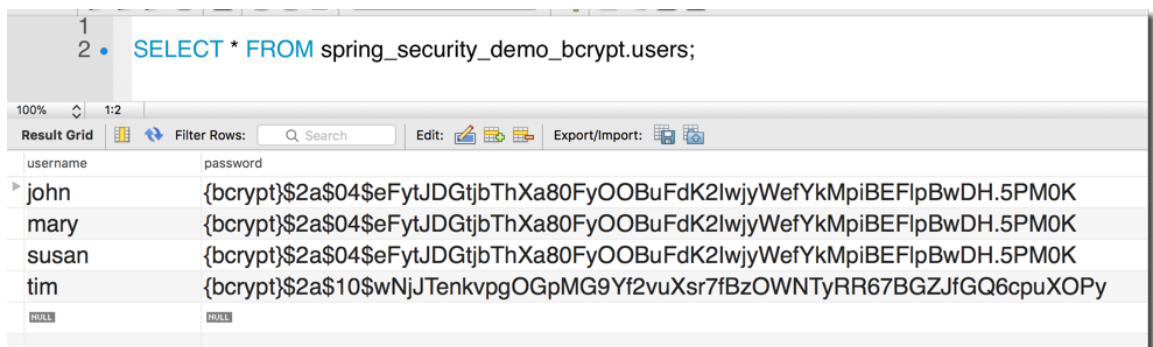
[Logout](#)

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.



The screenshot shows the MySQL Workbench interface. The SQL editor at the top contains the query: `SELECT * FROM spring_security_demo_bcrypt.users;`. Below the editor, the 'Result Grid' tab is active, displaying the query results in a table with two columns: 'username' and 'password'. The results show four users: john, mary, susan, and tim. Each user's password is stored as a bcrypt hash. The 'tim' user's password hash is longer than the others, indicating it was just added. There is also a row for 'NULL'.

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!