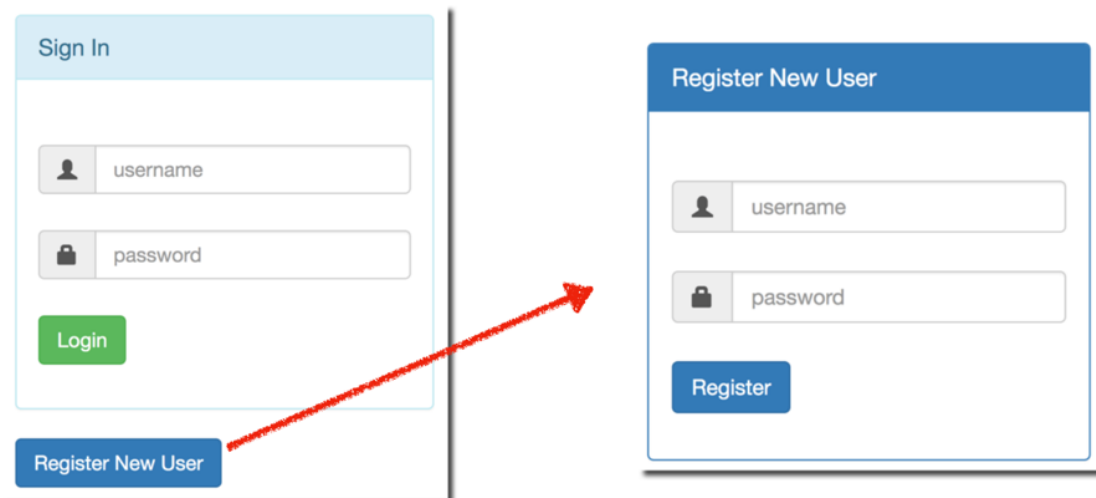# Spring Security
User Registration Tutorial

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-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. Add validation support to Maven POM
4. Create a CRM User class
5. Create a JDBC User Details Manager bean
6. Add button to login page for "Register New User"
7. Create Registration Form JSP
8. Create Registration Controller
9. Create Confirmation JSP
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 Eclipse, select **Import > Existing Maven Projects ...**
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: contains the database configuration file
- /src/main/webapp: contains the web files (jsp, css etc)
- /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. Add 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`

```
…
<!-- Hibernate Validator -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.7.Final</version>
</dependency>
…
```

4. Create a CRM User class

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

File: /src/main/java/com/luv2code/springsecurity/demo/user/CrmUser.java

```
package com.luv2code.springsecurity.demo.user;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class CrmUser {

        @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. Create a JDBC User Details Manager bean

In our security configuration file, `DemoSecurityConfig.java`, we create a JDBC User Details Manager bean. This is based on our security datasource. It provides access to the database for creating users. We'll also use `JdbcUserDetailsManager` to check if a user exists.

The `JdbcUserDetailsManager` has all of the low-level JDBC code for accessing the security database. There is no need for us to create the JDBC code ... `JdbcUserDetailsManager` will handle it for us ☺

File: /src/main/java/com/luv2code/springsecurity/demo/config/DemoSecurityConfig.java

```
…
@Bean
public UserDetailsManager userDetailsManager() {

        JdbcUserDetailsManager jdbcUserDetailsManager = new JdbcUserDetailsManager();

        jdbcUserDetailsManager.setDataSource(securityDataSource);

        return jdbcUserDetailsManager;
}
…
```
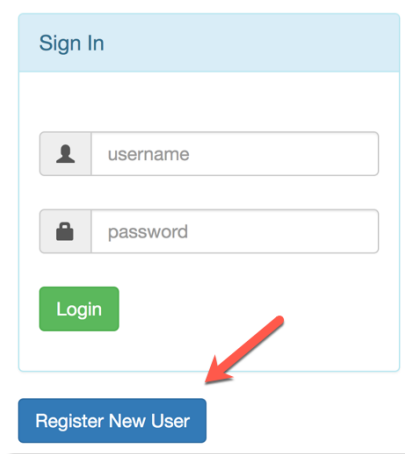
We'll use this bean later in the Registration Controller.

You can get more details on the `JdbcUserDetailsManager` class at the link below:

http://bit.ly/2oJHCO7

6. Add button to 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.
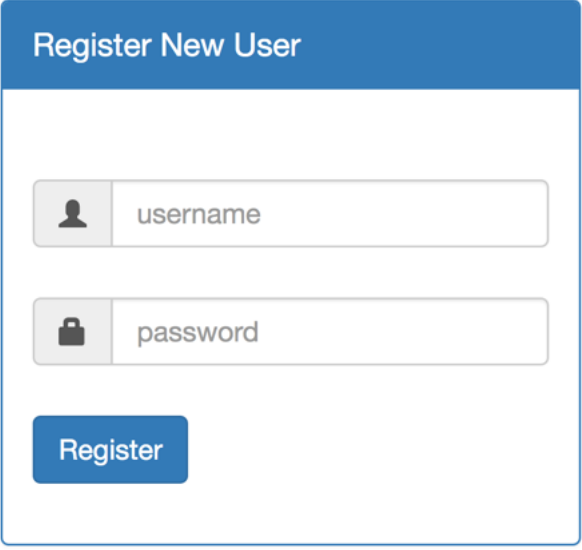
Near the bottom of the login form, see the new code.

File: /src/main/webapp/WEB-INF/view/fancy-login.jsp

```
<div>
    <a href="${pageContext.request.contextPath}/register/showRegistrationForm"
       class="btn btn-primary"
       role="button" aria-pressed="true">
       Register New User
    </a>
</div>
```

7. Create Registration Form JSP

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 `CrmUser`.

Below are the relevant snippets from the form.

File: /src/main/webapp/WEB-INF/view/registration-form.jsp

```
…
<!-- Registration Form -->
<form:form action="${pageContext.request.contextPath}/register/processRegistrationForm"
                modelAttribute="crmUser"
                class="form-horizontal">

        <!-- Check for registration error -->
        <c:if test="${registrationError != null}">
                <div class="alert alert-danger col-xs-offset-1 col-xs-10">
                        ${registrationError}
                </div>
        </c:if>
        …
        <!-- User name -->
        <form:input path="userName" placeholder="username" class="form-control" />

        <!-- Password -->
        <form:password path="password" placeholder="password" class="form-control" />

        <button type="submit" class="btn btn-primary">Register</button>

</form:form>
…
```

8. Create 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/springsecurity/demo/controller/RegistrationController.java

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

        ...

}
```

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

USERDETAILSMANAGER

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

```
        @Autowired
        private UserDetailsManager userDetailsManager;
```

Recall, this bean was created earlier in `DemoSecurityConfig.java`.

It is based on our security datasource. The `userDetailsManager` provides access to the database for creating users. We'll also use this bean to check if a user exists. The bean has all of the low-level JDBC code for accessing the security database. There is no need for us to develop the JDBC code ... this bean will handle it for us.

BCRYPTPASSWORDENCODER

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

```
        private PasswordEncoder passwordEncoder = new BCryptPasswordEncoder();
```

This code will create the `BCryptPasswordEncoder`. This class is part of the Spring Security framework. We will use it 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 `CrmUser` and add it as a model attribute.

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

        theModel.addAttribute("crmUser", new CrmUser());

        return "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:

```java
        @PostMapping("/processRegistrationForm")
        public String processRegistrationForm(
                        @Valid @ModelAttribute("crmUser") CrmUser theCrmUser,
                        BindingResult theBindingResult,
                        Model theModel) {

            // form validation

            // check the database if user already exists

            // encrypt the password

           // prepend the encoding algorithm id

            // give user default role of "employee"

            // create user details object

            // save user in the database

            return "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.

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

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

                return "registration-form";
        }
```

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

At the moment, our `CrmUser.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.

There is a helper method that does the work `doesUserExist(userName)`

```
    private boolean doesUserExist(String userName) {

        // check the database if the user already exists
        boolean exists = userDetailsManager.userExists(userName);

        return exists;
    }
```

This method makes use of the `userDetailsManager` 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

The first thing we need to do is encrypt the password.

```
    // encrypt the password
    String encodedPassword = passwordEncoder.encode(theCrmUser.getPassword());
```

We make use of the BCrypt password encoder created earlier.

The variable `theCrmUser` 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.

Next, we need to prepend the encoding algorithm id.

```
        // prepend the encoding algorithm id
        encodedPassword = "{bcrypt}" + encodedPassword;
```

> *Recall that in Spring Security, the passwords in the database have the form of:*
> *{encodingId}<<thePassword>>*

In our case of using BCrypt encryption it is: `{bcrypt}<<encodedPassword>>`

SET UP USER ROLES

Now we need to set up user roles. To keep things simple, in this example, the users have the role of Employee by default.

```
    // give user default role of "employee"
    List<GrantedAuthority> authorities =
                    AuthorityUtils.createAuthorityList("ROLE_EMPLOYEE");
```

USER

Next, we create the `User` object. The `User` is defined in the Spring Security Framework.

```
        // create user details object
        User tempUser = new User(userName, encodedPassword, authorities);
```

STORE USER IN DATABASE

Now, we are almost done. The final step is storing the user in the database.

```
        // save user in the database
        userDetailsManager.createUser(tempUser);
```

We make use of the `userDetailsManager` again. The method `createUser()` will handle all of the low-level JDBC work of adding the user and roles to the database.

RETURN CONFIRMATION PAGE

Once that is complete then we return to the registration confirmation page.

```
        return "registration-confirmation";
```

9. Create Confirmation JSP

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

`File: /src/main/webapp/WEB-INF/view/registration-confirmation.jsp`
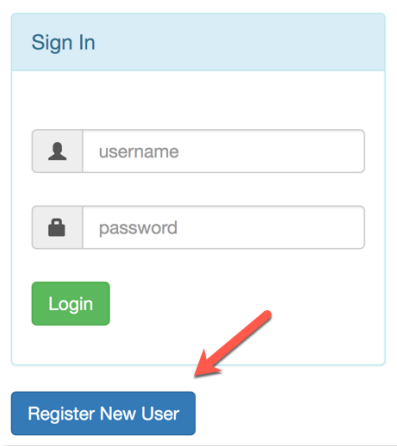
```
<body>

  <h2>User registered successfully!</h2>

  <hr>

  <a href="${pageContext.request.contextPath}/showMyLoginPage">Login with new user</a>

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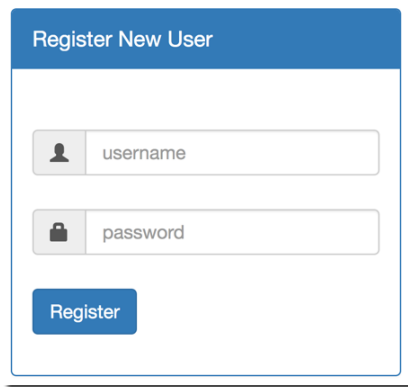
The user can now log in with the new account. ☺

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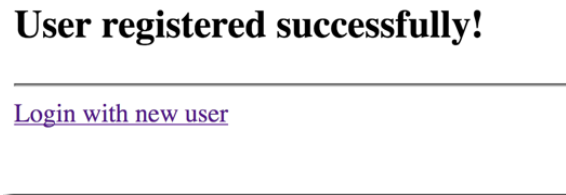
3. In the registration form, enter a new user name and password. For example:

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

4. Click the **Register** button.

   This will show the confirmation page.

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

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
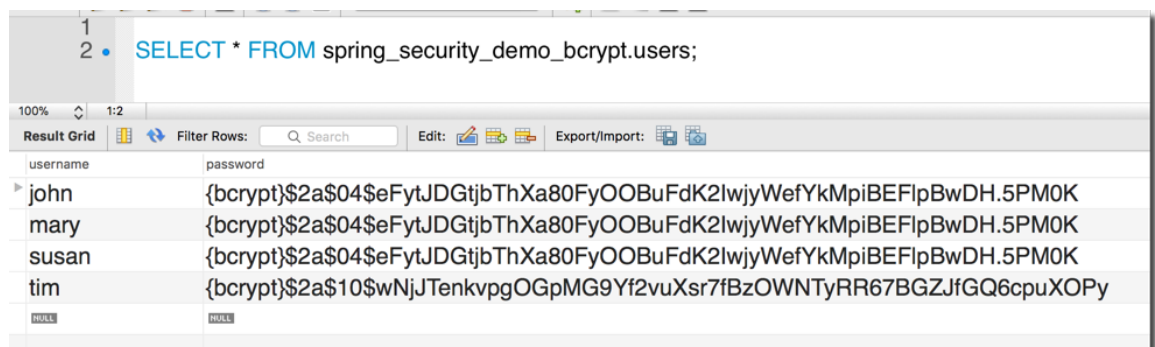
Welcome to the luv2code company home page!

User: tim

Role(s): [ROLE_EMPLOYEE]

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: **spring_security_demo_bcrypt**

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

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

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!