

Spring Security

User Registration - Custom User Details Tutorial

Introduction

In this tutorial, you will learn how to perform user registration - custom user details with Spring Security using Hibernate.

Custom user details include: user name, password, first name, last name and email.

We'll create a user registration form and store the user's information in the database.

The diagram illustrates the user registration process. On the left is a 'Sign In' form with fields for 'username' and 'password', a green 'Login' button, and a blue 'Register New User' button at the bottom. A red arrow points from the 'Register New User' button to the 'Register New User' form on the right. The 'Register New User' form has a blue header and fields for 'username (*)', 'password (*)', 'confirm password (*)', 'first name (*)', 'last name (*)', and 'email (*)', followed by a blue 'Register' button.

Sign In

Login

Register New User

Register New User

Register

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. Add Spring Transactions, ORM and Hibernate support to Maven POM
5. Define a BCryptPasswordEncoder and DaoAuthenticationProvider beans
6. Create a CRM User class
7. Add Persistence Properties
8. Create Custom User Details entity classes (User, Role)
9. Add custom validations to the User entity fields like password and email.
10. Add button to login page for "Register New User"
11. Create Registration Form JSP
12. Create Registration Controller
13. Create Service and Dao classes
14. Create Confirmation JSP
15. Test the App
16. 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-custom-user-code>
2. Unzip the file.

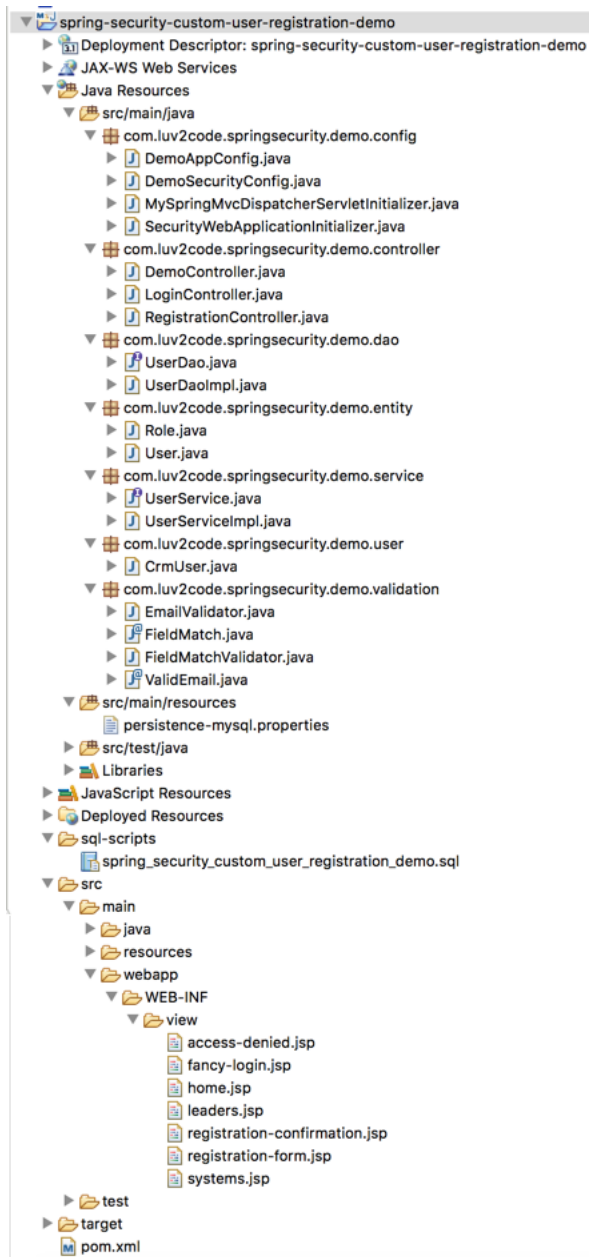
IMPORT THE CODE

1. In Eclipse, select **Import > Existing Maven Projects ...**
2. Browse to the 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

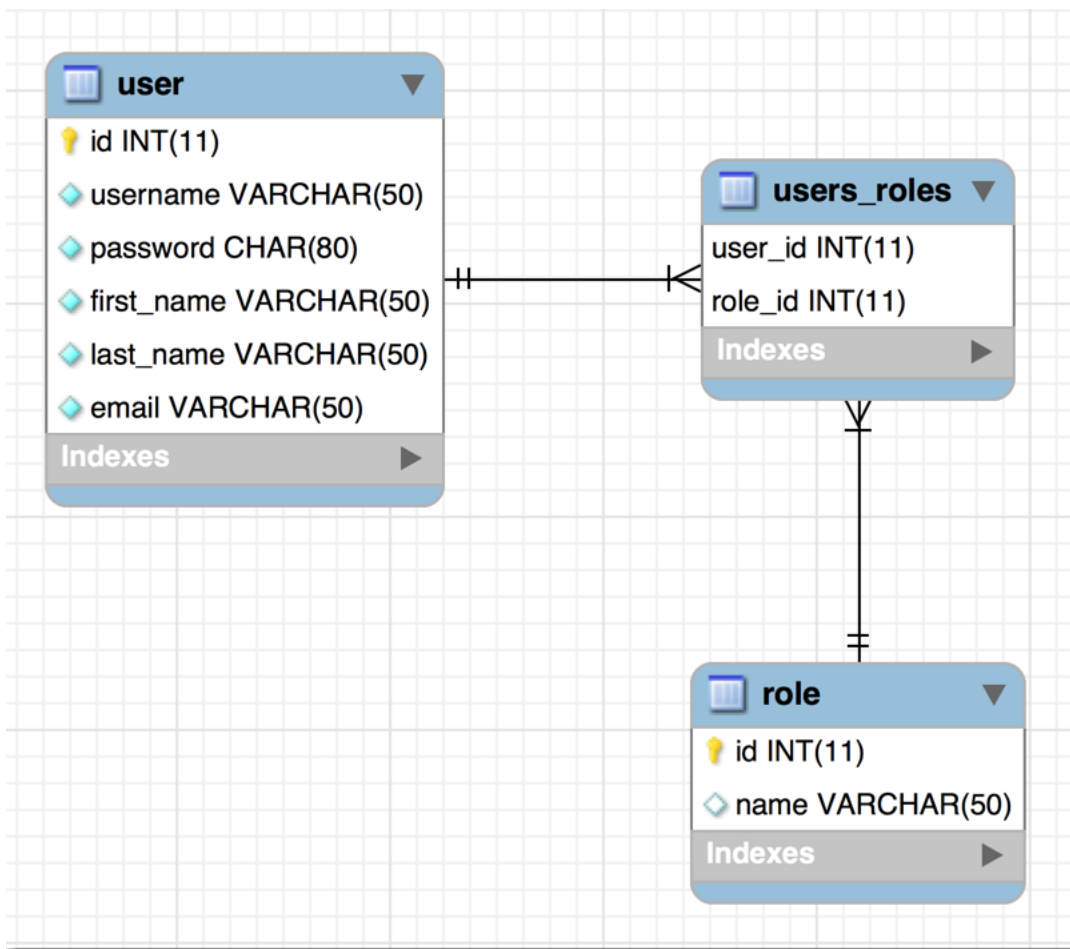
In order to make sure we are all on the same page in terms of database schema names and user accounts/passwords, let's run the same database scripts.

MYSQL WORKBENCH

In MySQL workbench, run the following database script:

```
/sql-scripts/spring_security_custom_user_registration_demo.sql
```

This script creates the database schema: **spring_security_custom_user_demo**.



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 validation rules to the registration form to make sure the fields are not empty, password matching and email validations.

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. Add Spring Transactions, ORM and Hibernate support to Maven POM

As we are implementing the custom user details in the registration, we are going to use the custom User and Role entities and save the user details to the database using the Hibernate. So, we need to add these dependencies in the pom.xml file.

```
<!-- Spring Transactions -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-tx</artifactId>
  <version>${springframework.version}</version>
</dependency>

<!-- Spring ORM -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-orm</artifactId>
  <version>${springframework.version}</version>
</dependency>

<!-- Hibernate Core -->
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>${hibernate.version}</version>
</dependency>
```

5. Define a BCryptPasswordEncoder and DaoAuthenticationProvider beans

In our security configuration file, DemoSecurityConfig.java, we define a BCryptPasswordEncoder and DaoAuthenticationProvider beans. We assign the UserService and PasswordEncoder to the DaoAuthenticationProvider.

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

```
...
//bcrypt bean definition
@Bean
public BCryptPasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}

//authenticationProvider bean definition
@Bean
public DaoAuthenticationProvider authenticationProvider() {
    DaoAuthenticationProvider auth = new DaoAuthenticationProvider();
    auth.setUserDetailsService(userService);
    auth.setPasswordEncoder(passwordEncoder());
    return auth;
}
...
```

We are assigning the custom user details and password encoder to the DaoAuthenticationProvider.

6. Create a CRM User class

For our registration form, we are creating a user class with custom details for the CRM project. It will have the username, password, first name, last name and email. 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;

import com.luv2code.springsecurity.demo.validation.FieldMatch;
import com.luv2code.springsecurity.demo.validation.ValidEmail;

@FieldMatch.List({
    @FieldMatch(first = "password", second = "matchingPassword", message = "The password
fields must match")
})
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;

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

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

    // other fields, constructor, getters/setters omitted for brevity
    ...
}
```


7. Add Persistence Properties

Add JDBC and Hibernate properties and define datastore bean in DemoAppConfig.java as discussed in CRM App.

File: /src/main/resources/persistence-mysql.properties

```
#
# JDBC connection properties
#
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/spring_security_custom_user_demo?useSSL=false
jdbc.user=hbstudent
jdbc.password=hbstudent

#
# Connection pool properties
#
connection.pool.initialPoolSize=5
connection.pool.minPoolSize=5
connection.pool.maxPoolSize=20
connection.pool.maxIdleTime=3000

#
# Hibernate properties
#
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.show_sql=true
hibernate.packagesToScan=com.luv2code.springsecurity.demo.model
```

8. Create Custom User Details entity classes (User, Role)

Creating the User and Role entity classes (We can use any name for these entities)

File: /src/main/java/com/luv2code/springsecurity/demo/entity/User.java

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

import javax.persistence.*;
import java.util.Collection;

@Entity
@Table(name = "user")
public class User {
```

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "id")
private Long id;

@Column(name = "username")
private String userName;

@Column(name = "password")
private String password;

@Column(name = "first_name")
private String firstName;

//other fields, constructor, getters/setters omitted for brevity
...
```

File: /src/main/java/com/luv2code/springsecurity/demo/entity/Role.java

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

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "role")
public class Role {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @Column(name = "name")
    private String name;

    //other fields, constructor, getters/setters omitted for brevity
}
```

9. Add custom validations to the User entity fields like password and email

We start with the email validator and make sure it's well-formed. We're going to be building a **custom validator** for that, as well as a **custom validation annotation** – let's call that `@ValidEmail`.

Here's the email validation annotation and the custom validator:

Custom Annotation for Email validation:

File: `/src/main/java/com/luv2code/springsecurity/demo/validation/ValidEmail.java`

```
@Constraint(validatedBy = EmailValidator.class)
@Target({ ElementType.TYPE, ElementType.FIELD, ElementType.ANNOTATION_TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface ValidEmail {
    String message() default "Invalid email";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

The Custom EmailValidator:

File: `/src/main/java/com/luv2code/springsecurity/demo/validation/EmailValidator.java`

```
public class EmailValidator implements ConstraintValidator<ValidEmail, String> {

    private Pattern pattern;
    private Matcher matcher;
    private static final String EMAIL_PATTERN = "^[_A-Za-z0-9-\\+](\\.[_A-Za-z0-9-\\+])*"
        + "[A-Za-z0-9-](\\.[A-Za-z0-9-])*(\\.[A-Za-z]{2,})$";

    @Override
    public boolean isValid(final String email, final ConstraintValidatorContext context) {
        pattern = Pattern.compile(EMAIL_PATTERN);
        if (email == null) {
            return false;
        }
        matcher = pattern.matcher(email);
        return matcher.matches();
    }
}
```

```
}  
  
}
```

We also need a custom annotation and validator to make sure that the password and matchingPassword fields match up. Here, we are using the generic field matching validator. We can use this for multiple fields in an array format.

Custom Annotation for Validating two fields:

File: /src/main/java/com/luv2code/springsecurity/demo/validation/FieldMatch.java

```
@Constraint(validatedBy = FieldMatchValidator.class)  
@Target({ ElementType.TYPE, ElementType.ANNOTATION_TYPE })  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface FieldMatch {  
    String message() default "";  
    Class<?>[] groups() default {};  
    Class<? extends Payload>[] payload() default {};  
  
    String first();  
    String second();  
  
    @Target({ ElementType.TYPE, ElementType.ANNOTATION_TYPE })  
    @Retention(RetentionPolicy.RUNTIME)  
    @Documented  
    @interface List  
    {  
        FieldMatch[] value();  
    }  
}
```

The Custom FieldMatchValidator:

File: /src/main/java/com/luv2code/springsecurity/demo/validation/FieldMatchValidator.java

```
public class FieldMatchValidator implements ConstraintValidator<FieldMatch, Object> {  
  
    private String firstFieldName;  
    private String secondFieldName;  
    private String message;  
  
    @Override
```

```
public void initialize(final FieldMatch constraintAnnotation) {
    firstFieldName = constraintAnnotation.first();
    secondFieldName = constraintAnnotation.second();
    message = constraintAnnotation.message();
}

@Override
public boolean isValid(final Object value, final ConstraintValidatorContext context)
{
    boolean valid = true;
    try
    {
        final Object firstObj = new
        BeanWrapperImpl(value).getPropertyValue(firstFieldName);
        final Object secondObj = new
        BeanWrapperImpl(value).getPropertyValue(secondFieldName);

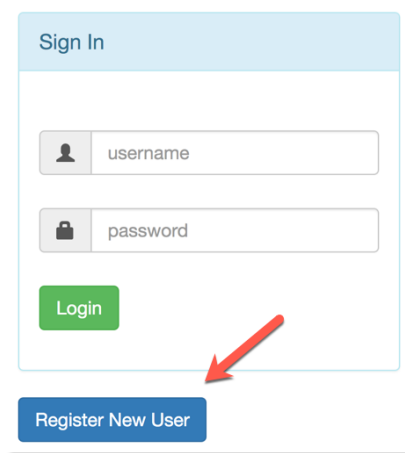
        valid = firstObj == null && secondObj == null || firstObj != null &&
        firstObj.equals(secondObj);
    }
    catch (final Exception ignore)
    {
        // we can ignore
    }

    if (!valid){
        context.buildConstraintViolationWithTemplate(message)
            .addPropertyNode(firstFieldName)
            .addConstraintViolation()
            .disableDefaultConstraintViolation();
    }

    return valid;
}
}
```

10. 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>
```

11. 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" />

    <!-- First Name -->
    <form:input path="firstName" placeholder="first name" class="form-control" />

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

</form:form>
...
```


12. 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.

USERSERVICE

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

```
@Autowired
private UserService userService;
```

This is nothing but the service interface which extends `UserDetailsService`. We are using this to find the user with the username and to create the user.

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, password, matching password, first name, last name, email. 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("crmUser") CrmUser theCrmUser,
    BindingResult theBindingResult,
    Model theModel) {

    // form validation

    // check the database if user already exists

    // 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.

```
// form validation
if (theBindingResult.hasErrors()) {
    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 fields, password and matching password matcher, email validations. 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.

```
User existing = userService.findByUserName(username);
If(existing!=null){
theModel.addAttribute("crmUser", new CrmUser());
theModel.addAttribute("registrationError", "User name already exists.");

return "registration-form";
```

This code checks the database to see if the user already exists (actual checking is done at Dao). Of course, we don't want to add users with same username.

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

STORE USER IN DATABASE

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

```
// create user account
userService.save(theCrmUser);
```

We make use of the `userService` again. We are using the Hibernate to save the user.

RETURN CONFIRMATION PAGE

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

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

13. Create Service and Dao classes

As discussed above, The `processRegistrationForm()` method in the Controller is the main focus of this bonus lecture. These are the two service/dao calls which do all the work and both the method names are self-explanatory.

```
userService.findByUserName(userName)
userService.save(theCrmUser)
```

The `UserService` extends `UserDetailsService`.

File: `/src/main/java/com/luv2code/springsecurity/demo/service/UserService.java`

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

import com.luv2code.springsecurity.demo.entity.User;
import com.luv2code.springsecurity.demo.user.CrmUser;
import org.springframework.security.core.userdetails.UserDetailsService;

public interface UserService extends UserDetailsService {

    User findByUserName(String userName);

    void save(CrmUser crmUser);

}
```

In the `UserServiceImpl` we implement the methods to lookup a user by username and save the user registration using the `CrmUser`. And make sure that we need to encode the password before saving the user.

File: `/src/main/java/com/luv2code/springsecurity/demo/service/UserServiceImpl.java`

```
@Service
public class UserServiceImpl implements UserService {

    // need to inject user dao
    @Autowired
    private UserDao userDao;
    @Autowired
    private BCryptPasswordEncoder passwordEncoder;
    @Override
    @Transactional
    public User findByUserName(String userName) {
```

```
// check the database if the user already exists
return userDao.findByUserName(userName);
}

@Override
@Transactional
public void save(CrmUser crmUser) {
    User user = new User();
    // assign user details to the user object
    user.setUserName(crmUser.getUserName());
    user.setPassword(passwordEncoder.encode(crmUser.getPassword()));
    user.setFirstName(crmUser.getFirstName());
    user.setLastName(crmUser.getLastName());
    user.setEmail(crmUser.getEmail());
    // give user default role of "employee"
    user.setRoles(Arrays.asList(new Role("ROLE_EMPLOYEE")));
    // save user in the database
    userDao.save(user);
}

@Override
@Transactional
public UserDetails loadUserByUsername(String userName) throws
UsernameNotFoundException {
    User user = userDao.findByUserName(userName);
    if (user == null) {
        throw new UsernameNotFoundException("Invalid username or
password.");
    }
    return new
org.springframework.security.core.userdetails.User(user.getUserName(),
user.getPassword(),
        mapRolesToAuthorities(user.getRoles()));
}

private Collection<? extends GrantedAuthority>
mapRolesToAuthorities(Collection<Role> roles) {
    return roles.stream().map(role -> new
SimpleGrantedAuthority(role.getName())).collect(Collectors.toList());
}
}
```

The corresponding method calls in the Dao layer.

UserDao

File: /src/main/java/com/luv2code/springsecurity/demo/dao/UserDao.java

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

```
import com.luv2code.springsecurity.demo.entity.User;

public interface UserDao {

    User findByUserName(String userName);

    void save(User user);

}
```

UserDaoImpl

File: /src/main/java/com/luv2code/springsecurity/demo/dao/UserDaoImpl.java

```
@Repository
public class UserDaoImpl implements UserDao {

    // need to inject the session factory
    @Autowired
    private SessionFactory sessionFactory;

    @Override
    public User findByUserName(String theUserName) {
        // get the current hibernate session
        Session currentSession = sessionFactory.getCurrentSession();

        // now retrieve/read from database using username
        Query<User> theQuery = currentSession.createQuery("from User where "
            + "userName=:uName", User.class);
        theQuery.setParameter("uName", theUserName);
        User theUser = null;
        try {
            theUser = theQuery.getSingleResult();
        } catch (Exception e) {
            theUser = null;
        }

        return theUser;
    }

    @Override
    public void save(User theUser) {
```

```
// get current hibernate session
Session currentSession = sessionFactory.getCurrentSession();

// create the user ... finally LOL
currentSession.saveOrUpdate(theUser);

}

}
```

14. 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

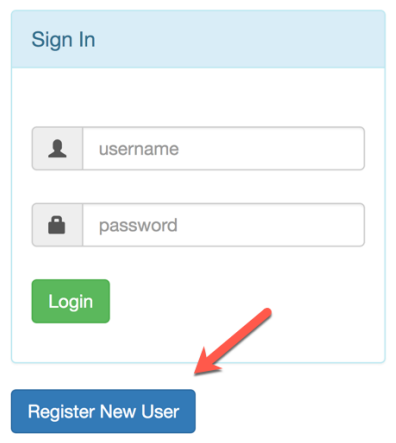
```
<body>
  <h2>User registered successfully!</h2>
  <hr>
  <a href="${pageContext.request.contextPath}/showMyLoginPage">Login with new user</a>
</body>
```

The user can now log in with the new account. 😊

15. Test the App

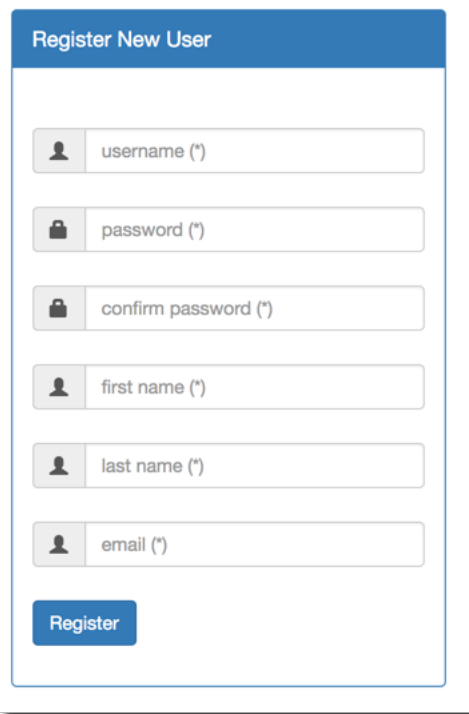
At this point, you can test the application.

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



The image shows a 'Sign In' form with a light blue header. It contains two input fields: 'username' with a person icon and 'password' with a lock icon. Below these is a green 'Login' button. At the bottom of the form is a blue 'Register New User' button. A red arrow points from the 'Login' button area down to the 'Register New User' button.

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



The image shows a 'Register New User' form with a blue header. It contains six input fields, each with an icon and an asterisk indicating it is required: 'username (*)' with a person icon, 'password (*)' with a lock icon, 'confirm password (*)' with a lock icon, 'first name (*)' with a person icon, 'last name (*)' with a person icon, and 'email (*)' with a person icon. At the bottom is a blue 'Register' button.

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

- username: **tim**
- password: **abc**
- confirm password: **abc**
- first name: **Tim**
- last name: **Public**
- email: **tim.pub@luv2code.com**

4. Click the **Register** button.

This will show the confirmation page.

User registered successfully!

[Login with new user](#)

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]

First name: Tim, Last name: Public, Email: tim.pub@luv2code.com

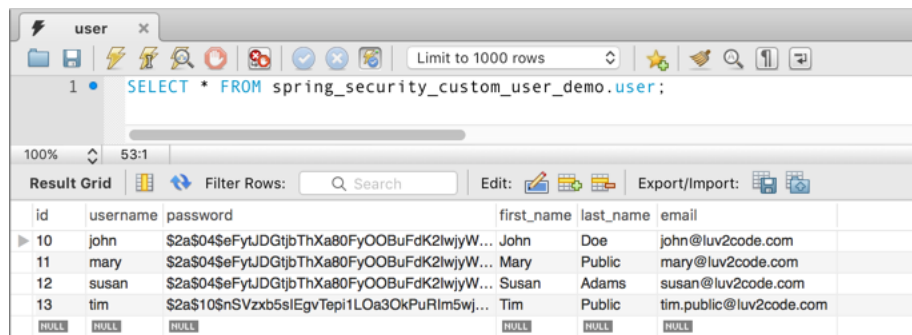
[Logout](#)

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

16. 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_custom_user_demo**
3. View the list of users in the **user** table.
4. You should see your new user along with their encrypted password and other details.



The screenshot shows the MySQL Workbench interface. The 'user' table is selected in the 'spring_security_custom_user_demo' schema. The SQL query 'SELECT * FROM spring_security_custom_user_demo.user;' is entered in the query editor. The 'Result Grid' shows the following data:

id	username	password	first_name	last_name	email
10	john	\$2a\$04\$eFytJDgtjbThXa80FyOOBuFdK2lwjyW...	John	Doe	john@luv2code.com
11	mary	\$2a\$04\$eFytJDgtjbThXa80FyOOBuFdK2lwjyW...	Mary	Public	mary@luv2code.com
12	susan	\$2a\$04\$eFytJDgtjbThXa80FyOOBuFdK2lwjyW...	Susan	Adams	susan@luv2code.com
13	tim	\$2a\$10\$SnVzxb5slEgvTepi1LOa3OkPuRlm5wj...	Tim	Public	tim.public@luv2code.com

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