Section 47: Spring Security - Getting Started

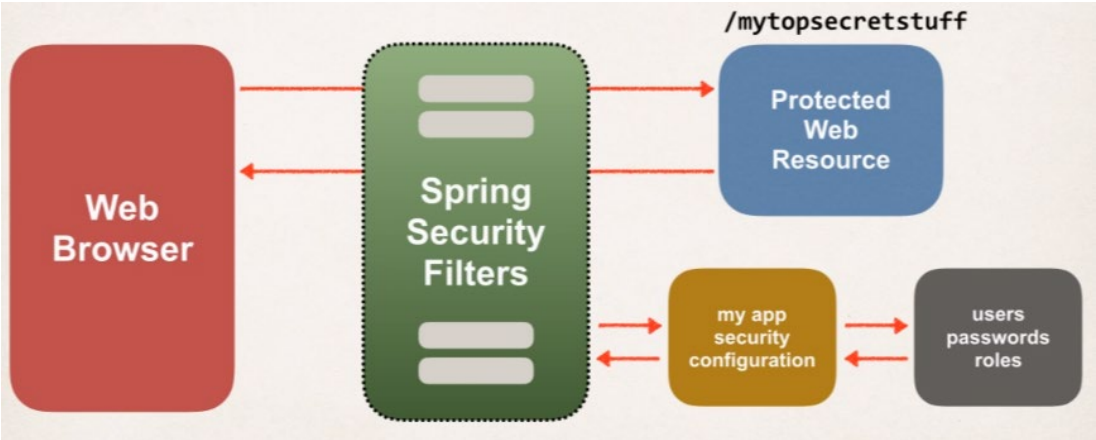
0 / 18 | 1hr 4min

Overview

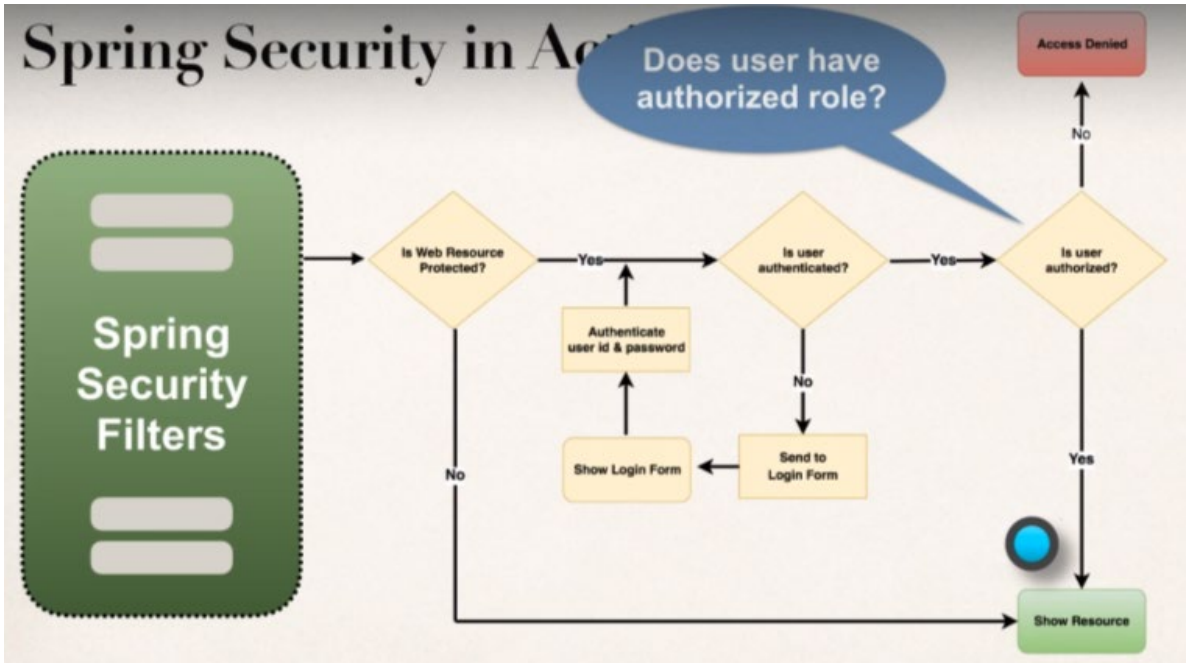
- Spring security defines a framework for security
- Implemented using Servlet filters in the background.
- There are two methods of securing a web application: Declarative and programmatic

Servlet Filters are user to pre-process / post-process web requests.

Servlet Filters can route web requests based on security logic.



Spring Security filters will intercept the incoming requests, pre-process them and see whether the user can actually access the protected web resource; so, it will look into the app's security configuration and also look into users, passwords, roles to authenticate the user and also authorize to access the web resource.



Declarative Security

- Define application's security constraints in configuration
  - All Java config (@Configuration, no xml)
  - or Spring XML config
- Provides separation of concerns between application code and security

Programmatic Security

- Spring Security provides an API for custom application coding
- Provides greater customization for specific app requirements

Different Login Methods:

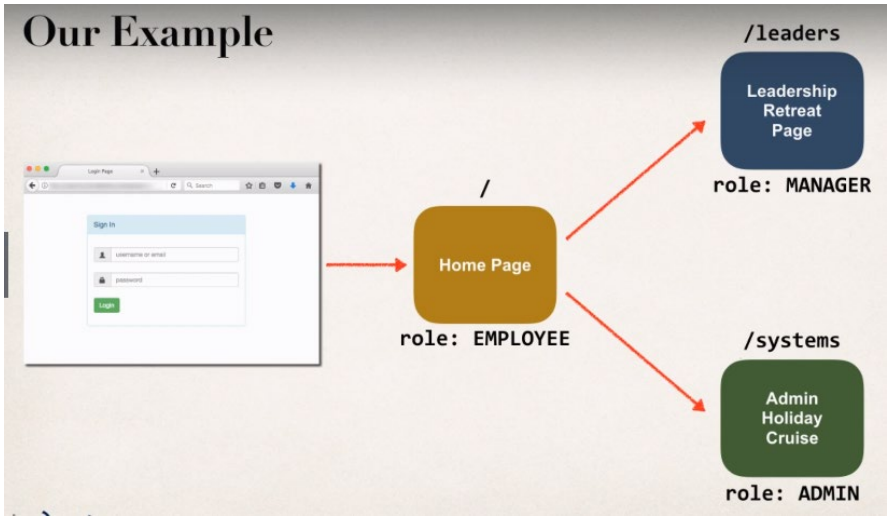
1. Basic HTTP Authentication
2. Default Login Form

- Spring Security provides a default login form.

3. Custom login form

- your own look-and-feel, HTML+CSS.

Demo of our Example



Java Configuration

Development Process:

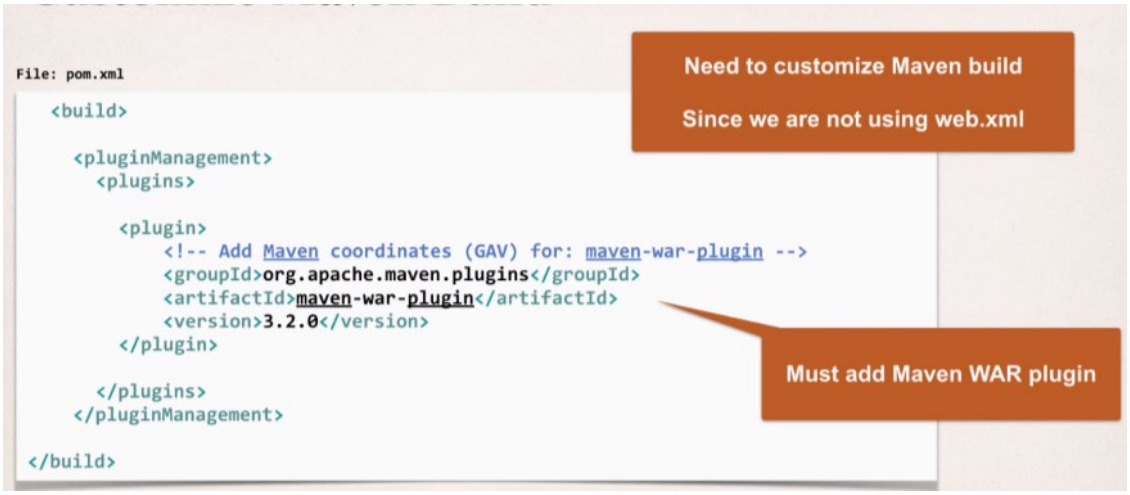
1. Add Maven Dependencies for Spring MVC Web App.

   So, we need all these Artifact ids in our **pom.xml** file.

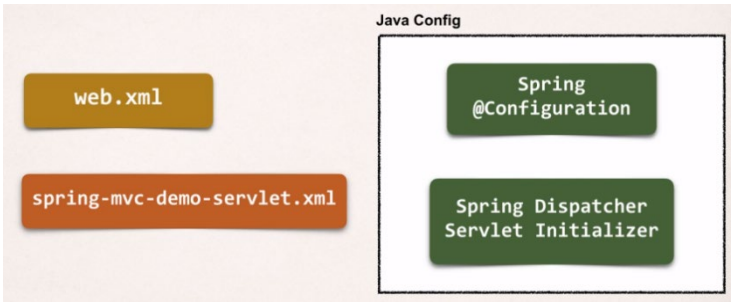
File: pom.xml

```
<!-- Spring MVC support -->
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-webmvc</artifactId>
   <version>...</version>
</dependency>

<!-- Servlet, JSP and JSTL support -->
<dependency>
   <groupId>javax.servlet</groupId>
   <artifactId>javax.servlet-api</artifactId>
   <version>...</version>
</dependency>

<dependency>
   <groupId>javax.servlet.jsp</groupId>
   <artifactId>javax.servlet.jsp-api</artifactId>
   <version>...</version>
</dependency>

<dependency>
   <groupId>javax.servlet</groupId>
   <artifactId>jstl</artifactId>
   <version>...</version>
</dependency>
```

And we need to customize the Maven Build, because we are not using web.xml and here Maven will actually complain that our project is not having web.xml, so we will make use of maven-war-plugin, and we will use this plugin to tell maven that we are using pure java configuration for running our application (no XML). So, this way we are customizing the maven build process.



Now in place of web.xml and spring-mvc-demo-servlet.xml that we have used earlier, we will use pure java configuration with no XML.



Flash back to XML config which we will convert into Java Code:

File: spring-mvc-demo-servlet.xml

Just an FYI

```
<beans>

  <!-- Add support for component scanning -->
  <context:component-scan base-package="com.luv2code.springdemo" />

  <!-- Add support for conversion, formatting and validation support -->
  <mvc:annotation-driven/>

  <!-- Define Spring MVC view resolver -->
  <bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/view/" />
    <property name="suffix" value=".jsp" />
  </bean>

</beans>
```

Enabling the MVC Java Config

@EnableWebMvc

- Provides similar support to `<mvc:annotation-driven />` in XML.
- Adds conversion, formatting and validation support
- Processing of `@Controller` classes and `@RequestMapping` etc ... methods

2. Create Spring App Configuration (@Configuration)

File: DemoAppConfig.java

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages="com.luv2code.springsecurity.demo")
public class DemoAppConfig {

  // define a bean for ViewResolver

  @Bean
  public ViewResolver viewResolver() {

    InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();

    viewResolver.setPrefix("/WEB-INF/view/");
    viewResolver.setSuffix(".jsp");

    return viewResolver;
  }

}
```

View Resolver Configs - Explained

```
InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();

viewResolver.setPrefix("/WEB-INF/view/");
viewResolver.setSuffix(".jsp");
```

/WEB-INF/view/show-student-list.jsp

View name

3. Create Spring Dispatcher Servlet Initializer

   Flash back to XML config of web.xml, and now we need to convert this into pure java based.

File: web.xml

Just an FYI

```
<web-app>

  <servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring-mvc-demo-servlet.xml</param-value>
    </init-param>

    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

</web-app>
```

Web App Initializer:

Spring MVC make use of a special class called the **`AbstractAnnotationConfigDispatcherServletInitializer`**.

- Spring MVC provides support for web app initialization
- Makes sure your code is automatically detected
- Your code is used to initialize the servlet container

`AbstractAnnotationConfigDispatcherServletInitializer`

Web App Initializer (more info)

`AbstractAnnotationConfigDispatcherServletInitializer`

- Your TO DO list
  - Extend this abstract base class
  - Override required methods
  - Specify servlet mapping and location of your app config

File:MySpringMvcDispatcherServletInitializer.java

```
import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class MySpringMvcDispatcherServletInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

   @Override
   protected Class<?>[] getRootConfigClasses() {
      // TODO Auto-generated method stub
      return null;
   }

   @Override
   protected Class<?>[] getServletConfigClasses() {
      return new Class[] { DemoAppConfig.class };
   }

   @Override
   protected String[] getServletMappings() {
      return new String[] { "/" };
   }

}
```

```
<servlet>
   <servlet-name>dispatcher</servlet-name>
   <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

   <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring-mvc-demo-servlet.xml</param-value>
   </init-param>

   <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
   <servlet-name>dispatcher</servlet-name>
   <url-pattern>/</url-pattern>
</servlet-mapping>
```

4. Develop our Spring Controller.

File: DemoController.java

```
@Controller
public class DemoController {

  @GetMapping("/")
  public String showHome() {

    return "home";
  }

}
```

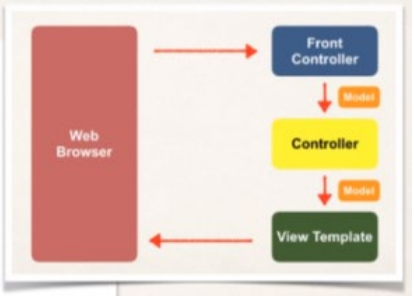
View name

/WEB-INF/view/home.jsp

5. Develop our JSP view Page.

File: /WEB-INF/view/home.jsp

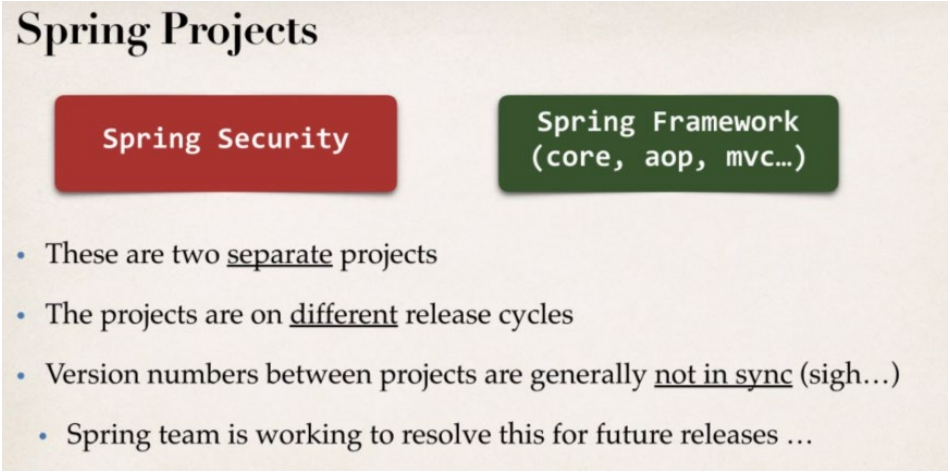
```
<html>

<body>

  Welcome to the luv2code company home page!

</body>

</html>
```



Spring Security Project Setup:

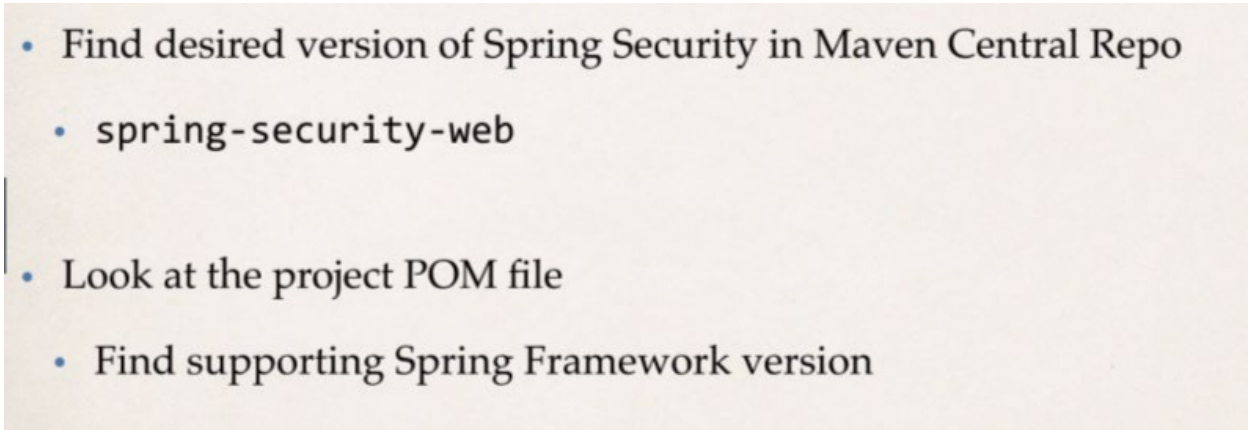
Add Maven dependencies for Spring Security

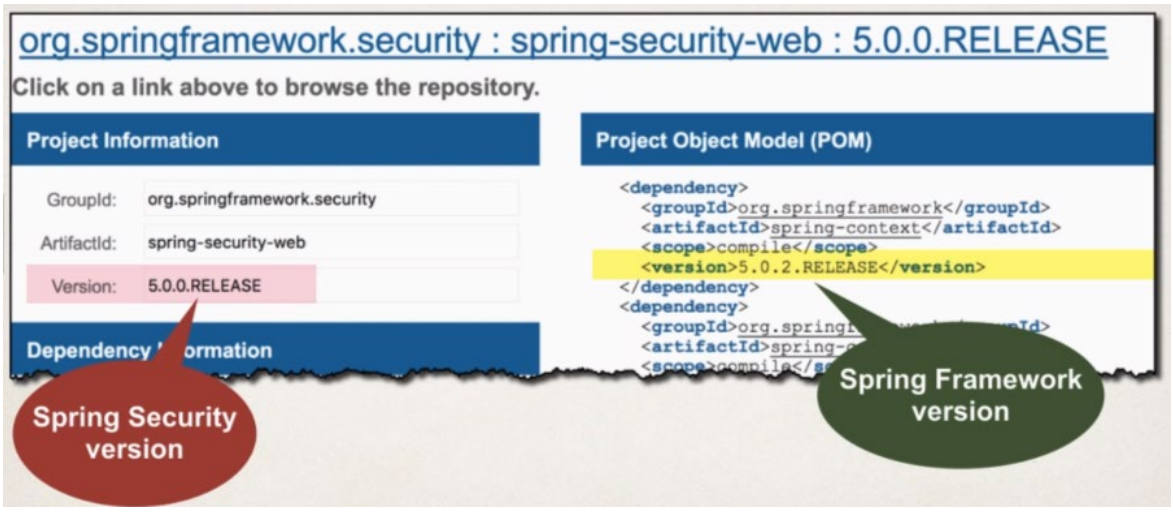
- Spring Security has two dependencies
  - spring-security-web
  - spring-security-config



- ✓ **Inhere, the common pitfall is using incompatible projects**
- ✓ **Need to find the compatible version**

Finding Compatible version:





So, we in order to add the Spring security dependencies in pom.xml, we will add a property called `springsecurity.version` then we will drop in reference to the spring framework and then we will drop in reference to spring security (spring-security-web, spring-security-config).

Make sure we use compatible versions otherwise we may have unpredictable results.

File: pom.xml

```
<properties>
    <springframework.version>5.0.2.RELEASE</springframework.version>
    <springsecurity.version>5.0.0.RELEASE</springsecurity.version>
</properties>

...

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

    <!-- Spring Security -->
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-web</artifactId>
        <version>${springsecurity.version}</version>
    </dependency>

    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-config</artifactId>
        <version>${springsecurity.version}</version>
    </dependency>
```

Spring Security - Latest Version

UPDATES - Spring Security 5.4.0
Spring Security 5.4.0 was released on 10 September 2020. This is a maintenance/bug fix release.

Spring Security 5.4.0 is compatible with Spring Framework 5.2.8

If you want to use the latest version, update your Maven POM file to use the compatible versions below

```
<springframework.version>5.2.8.RELEASE</springframework.version>

<springsecurity.version>5.4.0.RELEASE</springsecurity.version>
```

- **UPDATES - Spring Security 5.2.1**

Spring Security 5.2.1 was released on 4 November 2019. This is a maintenance/bug fix release.

Spring Security 5.2.1 is compatible with Spring Framework 5.2.1

If you want to use the latest version, update your Maven POM file to use the compatible versions below

```
<springframework.version>5.2.1.RELEASE</springframework.version>
```

```
2.
3.  <springsecurity.version>5.2.1.RELEASE</springsecurity.version>
```

- **UPDATES - Spring Security 5.2.0**

Spring Security 5.2.0 was released on 1 October 2019. This is a maintenance/bug fix release.

Spring Security 5.2.0 is compatible with Spring Framework 5.2.0

If you want to use the latest version, update your Maven POM file to use the compatible versions below

```
1.  <springframework.version>5.2.0.RELEASE</springframework.version>
2.
3.  <springsecurity.version>5.2.0.RELEASE</springsecurity.version>
```

- **UPDATES - Spring Security 5.1.6**

Spring Security 5.1.6 was released on 5 August 2019. This is a maintenance/bug fix release.

Spring Security 5.1.6 is compatible with Spring Framework 5.1.9

If you want to use the latest version, update your Maven POM file to use the compatible versions below

```
1.  <springframework.version>5.1.9.RELEASE</springframework.version>
2.
3.  <springsecurity.version>5.1.6.RELEASE</springsecurity.version>
```

- **UPDATES - Spring Security 5.1.5**

Spring Security 5.1.5 was released on 2 April 2019. This is a maintenance/bug fix release.

Spring Security 5.1.5 is compatible with Spring Framework 5.1.6

If you want to use the latest version, update your Maven POM file to use the compatible versions below

```
1.  <springframework.version>5.1.6.RELEASE</springframework.version>
2.
3.  <springsecurity.version>5.1.5.RELEASE</springsecurity.version>
```

- **UPDATES - Spring Security 5.1.4**

Spring Security 5.1.4 was released on 14 February 2019. This is a maintenance/bug fix release.

Spring Security 5.1.4 is compatible with Spring Framework 5.1.5

If you want to use the latest version, update your Maven POM file to use the compatible versions below

```
1.  <springframework.version>5.1.5.RELEASE</springframework.version>
2.
3.  <springsecurity.version>5.1.4.RELEASE</springsecurity.version>
```

- UPDATES - Spring Security 5.1.3
Spring Security 5.1.3 was released on 10 January 2019. This is a maintenance/bug fix release.

Spring Security 5.1.3 is compatible with Spring Framework 5.1.4

If you want to use the latest version, update your Maven POM file to use the compatible versions below

```
1. <springframework.version>5.1.4.RELEASE</springframework.version>
2.
3. <springsecurity.version>5.1.3.RELEASE</springsecurity.version>
```

- **UPDATES - Spring Security 5.1.2**

Spring Security 5.1.2 was released on 29 November 2018. This is a maintenance/bug fix release.

Spring Security 5.1.2 is compatible with Spring Framework 5.1.3

If you want to use the latest version, update your Maven POM file to use the compatible versions below

```
1. <springframework.version>5.1.3.RELEASE</springframework.version>
2.
3. <springsecurity.version>5.1.2.RELEASE</springsecurity.version>
```

- **UPDATES - Spring Security 5.1.1**
Spring Security 5.1.1 was released on 16 October 2018. This is a maintenance/bug fix release.

Spring Security 5.1.1 is compatible with Spring Framework 5.1.1

If you want to use the latest version, update your Maven POM file to use the compatible versions below

```
1. <springframework.version>5.1.1.RELEASE</springframework.version>
2.
3. <springsecurity.version>5.1.1.RELEASE</springsecurity.version>
```

- **UPDATES - Spring Security 5.1.0**
Spring Security 5.1.0 was released on 27 September 2018. This is a maintenance/bug fix release.

Spring Security 5.1.0 is compatible with Spring Framework 5.1.0

If you want to use the latest version, update your Maven POM file to use the compatible versions below

```
1. <springframework.version>5.1.0.RELEASE</springframework.version>
2.
3. <springsecurity.version>5.1.0.RELEASE</springsecurity.version>
```

- **UPDATES - Spring Security 5.0.7**

Spring Security 5.0.7 was released on 26 July 2018. This is a maintenance/bug fix release.

Spring Security 5.0.7 is compatible with Spring Framework 5.0.8

If you want to use the latest version, update your Maven POM file to use the compatible versions below

```
<springframework.version>5.0.8.RELEASE</springframework.version>

<springsecurity.version>5.0.7.RELEASE</springsecurity.version>
```

- **UPDATES - Spring Security 5.0.6**

Spring Security 5.0.6 was released on 13 June 2018. This is a maintenance/bug fix release.

Spring Security 5.0.6 is compatible with Spring Framework 5.0.7

If you want to use the latest version, update your Maven POM file to use the compatible versions below

```
<springframework.version>5.0.7.RELEASE</springframework.version>

<springsecurity.version>5.0.6.RELEASE</springsecurity.version>
```

- **UPDATES - Spring Security 5.0.5**

Spring Security 5.0.5 was released on 8 May 2018. This is a maintenance/bug fix release.

Spring Security 5.0.5 is compatible with Spring Framework 5.0.6

If you want to use the latest version, update your Maven POM file to use the compatible versions below

```
<springframework.version>5.0.6.RELEASE</springframework.version>

<springsecurity.version>5.0.5.RELEASE</springsecurity.version>
```

- **UPDATES - Spring Security 5.0.4**

Spring Security 5.0.4 was released on 5 April 2018. This is a maintenance/bug fix release.

Spring Security 5.0.4 is compatible with Spring Framework 5.0.5

If you want to use the latest version, update your Maven POM file to use the compatible versions below

```
<springframework.version>5.0.5.RELEASE</springframework.version>

<springsecurity.version>5.0.4.RELEASE</springsecurity.version>
```

- **UPDATES - Spring Security 5.0.3**

Spring Security 5.0.3 was released the week of 28 Feb 2018. This is a maintenance/bug fix release.

Spring Security 5.0.3 is compatible with Spring Framework 5.0.4

If you want to use the latest version, update your Maven POM file to use the compatible versions below

```
<springframework.version>5.0.4.RELEASE</springframework.version>

<springsecurity.version>5.0.3.RELEASE</springsecurity.version>
```

- **UPDATES - Spring Security 5.0.2**

Spring Security 5.0.2 was released the week of 20 Feb 2018. This is a maintenance/bug fix release.

Spring Security 5.0.2 is compatible with Spring Framework 5.0.4

If you want to use the latest version, update your Maven POM file to use the compatible versions below

```
<springframework.version>5.0.4.RELEASE</springframework.version>

<springsecurity.version>5.0.2.RELEASE</springsecurity.version>
```

- **UPDATES Spring Security 5.0.1**

Spring Security 5.0.1 was released the week of 24 Jan 2018. This is a maintenance/bug fix release.

Spring Security 5.0.1 is compatible with Spring Framework 5.0.3

If you want to use the latest version, update your Maven POM file to use the compatible versions below

```
<springframework.version>5.0.3.RELEASE</springframework.version>

<springsecurity.version>5.0.1.RELEASE</springsecurity.version>
```

Spring Security – Basic Security (Users, Passwords and Roles)

Development Process:

1. Create the Spring Security Initializer.
2. Also Create Spring Security Configuration (@Configuration)
3. Add users, passwords and roles.

Step 1: Create the Spring Security Initializer:

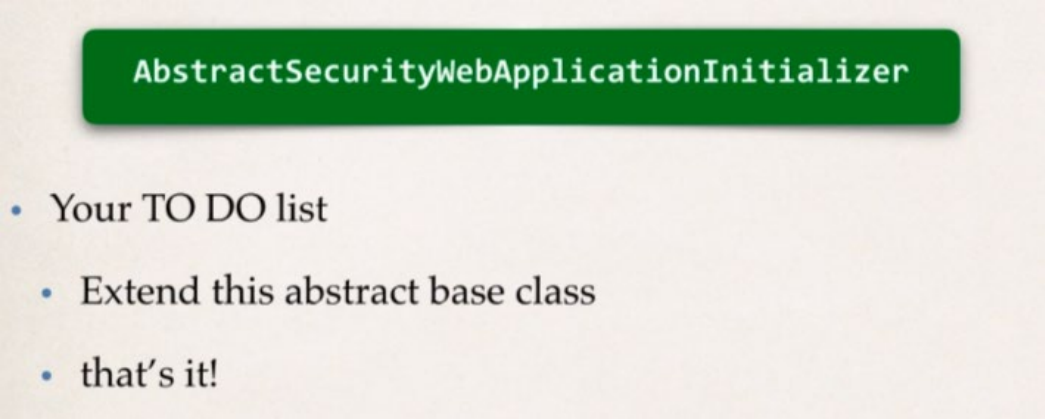
Spring Security Web App Initializer

Spring Security provides support for security initialization for initializing the security framework.

Our Security code is used to initialize the Servlet container.

The most important thing it will do is that it will register the spring security filters and make them active.

Special class is used to register **Spring Security Filters.** It's a very important class to have if we don't then spring security will not be registered, activated.



Sample Example of the class:

File: SecurityWebApplicationInitializer.java

```
import org.springframework.security.web.context.AbstractSecurityWebApplicationInitializer;

public class SecurityWebApplicationInitializer extends AbstractSecurityWebApplicationInitializer {

}
```

We don't need to override any method just as it's mentioned above.

Step 2: Create Spring Security Configuration:

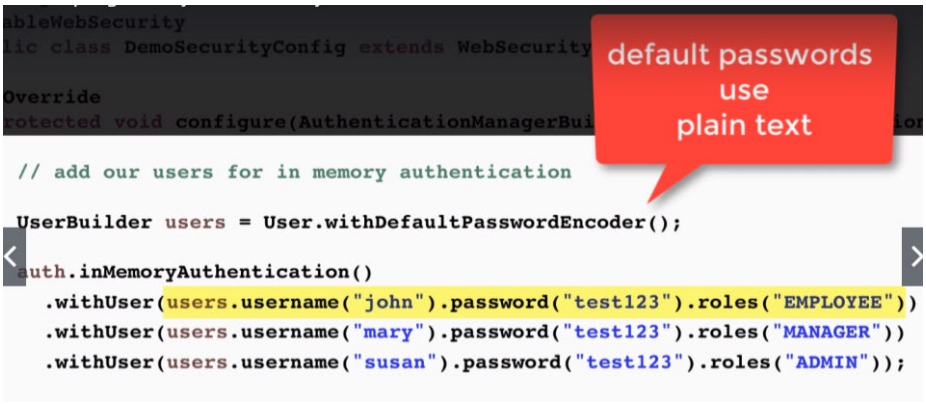
File: DemoSecurityConfig.java

```
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
public class DemoSecurityConfig extends WebSecurityConfigurerAdapter {

}
```

Step 3: Adding users, passwords and roles:

Here, we are going to keep user for in memory authentication, so we will hard code the users here for the security demo.

To add the users, we simply override the configure method and it will have parameter **(AuthenticationManagerBuilder auth**) from the Spring framework an we will use this to add the type of authentication we are going to use as in this example we will use in-memory authentication.



File: DemoSecurityConfig.java

```
@Configuration
@EnableWebSecurity
public class DemoSecurityConfig extends WebSecurityConfigurerAdapter {

  @Override
  protected void configure(AuthenticationManagerBuilder auth) throws Exception {

    // add our users for in memory authentication

    UserBuilder users = User.withDefaultPasswordEncoder();

    auth.inMemoryAuthentication()
      .withUser(users.username("john").password("test123").roles("EMPLOYEE"))
      .withUser(users.username("mary").password("test123").roles("MANAGER"))
      .withUser(users.username("susan").password("test123").roles("ADMIN"));

  }
```

When we do the authentication from the database then we will simply change the highlighted lines of code with appropriate piece of database information.

Once we are done with the above configurations and run the application then there will be a default login page on the welcome screen:

And once we enter the correct username and password that we have given in our in memory database then it will get us to our home page that we have configure into our controller.

Controller class:

```
SecurityWebA...   DemoSecurit...   Luv2code Com...   DemoControll...
1  package com.luv2code.springsecurity.demo.controller;
2
3  import org.springframework.stereotype.Controller;
5
6  @Controller
7  public class DemoController {
8
9      @GetMapping("/")
10     public String showHome() {
11         return "home";
12     }
13 }
14
```

Output: After entering the one of the correct credentials that we have configure above;

Section 48: Spring Security - Adding Custom Login Form

0 / 10 | 44min

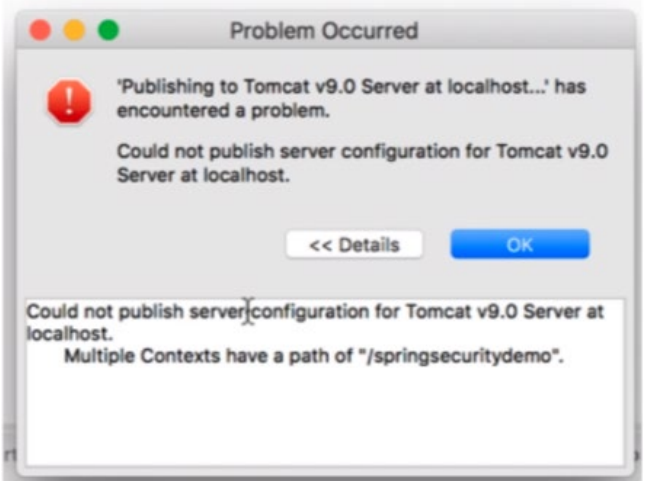
➢ Spring Security – Rename Context Roots

Basically, a Context root is the root path for our web application.

Suppose: we have a context root of **my-ecommerce-app** then we can access it at

http://localhost:8080/my-ecommerce-app

In case tomcat server having two web apps with same context root then it will throw **error**:

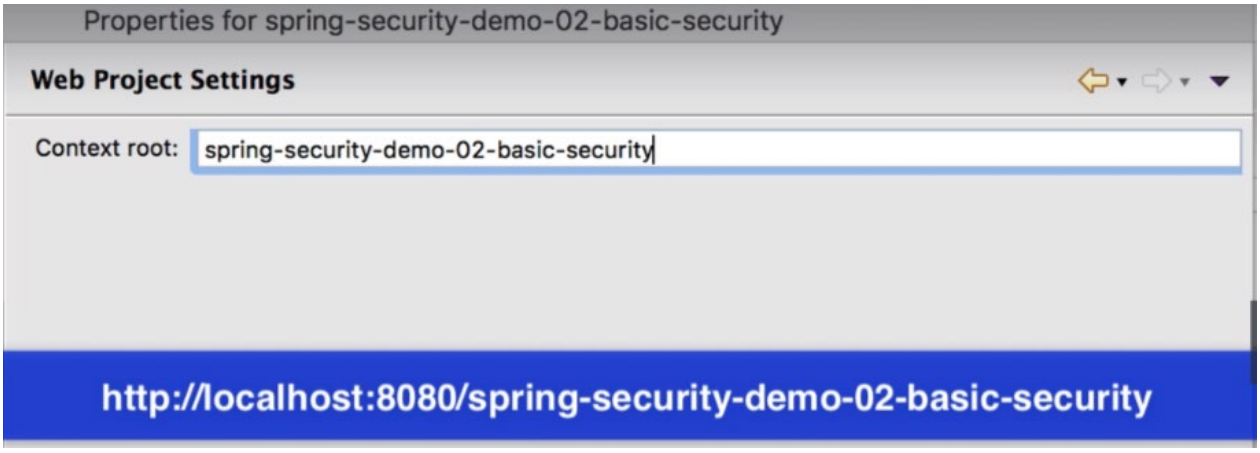


So, to resolve this, we need to keep the context roots for all the applications which are running in the tomcat server different.

So, to change the context root:

- Right click on **project**
- Go to **properties**
- Click on **Web Project Settings**

- In the web project settings **tab**: **change** the context root **textfield**
- Then hit apply and apply and close then hit OK to close all dialogs there.



Just give unique names to context root for all the projects and then we will be able to run different project at a time on the tomcat server.

➢ Spring Security – Custom Login Form Overview

So, we want to create our own custom login form with HTML+CSS, more control over how the form will look.

What we will do here is in the Spring Security filters we will setup the configuration i.e when the user login and we send the login form to the user then we will configure Spring to use our Custom login form.

Development Process:

1. Modify Spring Security Configuration to reference custom login form.

   So, we will override another method inside the **`DemoSecurityConfig`** where we have setup our inmemory users,

File: DemoSecurityConfig.java

```
@Configuration
@EnableWebSecurity
public class DemoSecurityConfig extends WebSecurityConfigurerAdapter {

   // override method to configure users for in-memory authentication …

}
```

Method	Description
`configure(AuthenticationManagerBuilder)`	Configure users (in memory, database, ldap, etc)
`configure(HttpSecurity)`	Configure security of web paths in application, login, logout etc

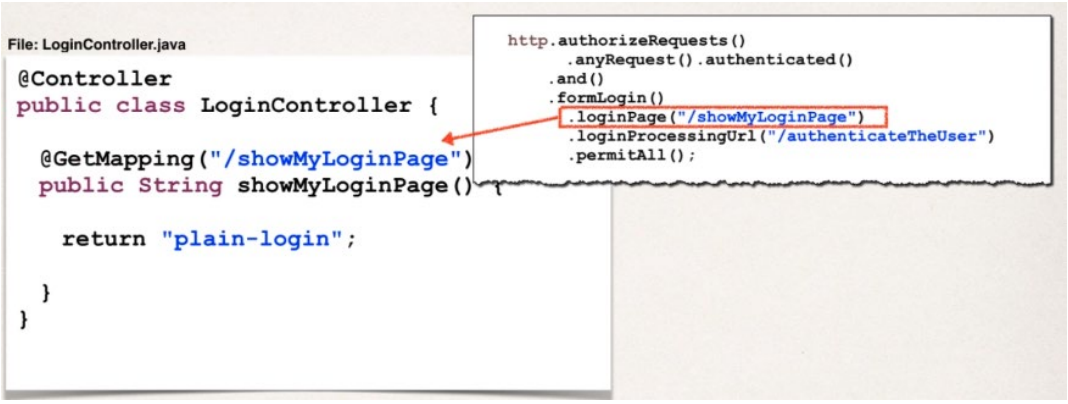
```
@Override
protected void configure(HttpSecurity http) throws Exception {

 http.authorizeRequests()
       .anyRequest().authenticated()
     .and()
     .formLogin()
       .loginPage("/showMyLoginPage")
       .loginProcessingUrl("/authenticateTheUser")
       .permitAll();

}
```

Line by line code explanation:

- This configure method has parameter ***HttpSecurity*** which we are using to configure spring security.
- First line ***authorizeRequests***() with which we will restrict access based on the servlet request coming in.
- Then ***.anyRequest().authenticated()*** means any request coming to the app must be authenticated i.e., user must be logged in.
- And for our ***formLogin(),*** we are customizing the login process and for the actual login page we will show our custom form at the request mapping ***"/showMyLoginPage"***
- Then login form will submit the data to the url ***"/authenticateTheUser"*** for processing and this is where spring security will go and check the userid and password.
- And ***permitAll()*** means we are allowing everyone to see the Login page

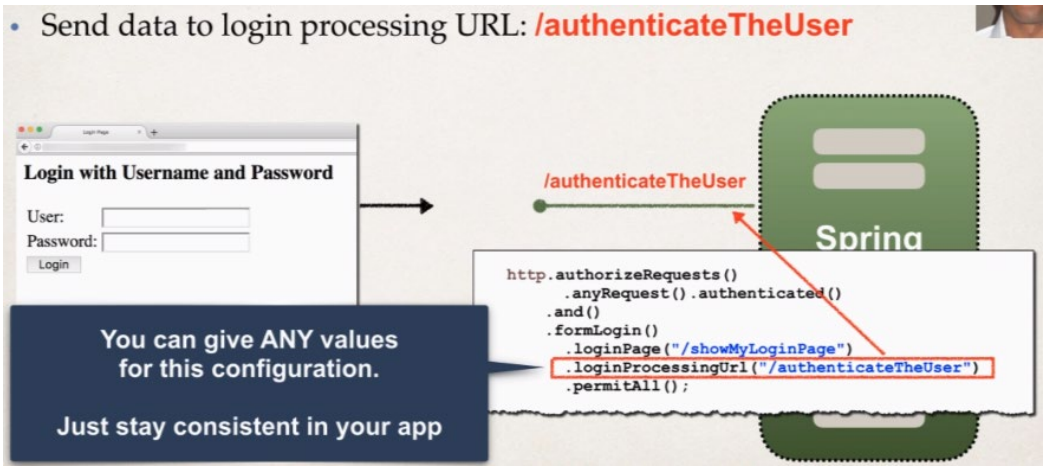
Now we need to create Controller for the request Mapping ***"/showMyLoginPage"***

2. Develop a Controller to show the custom login form

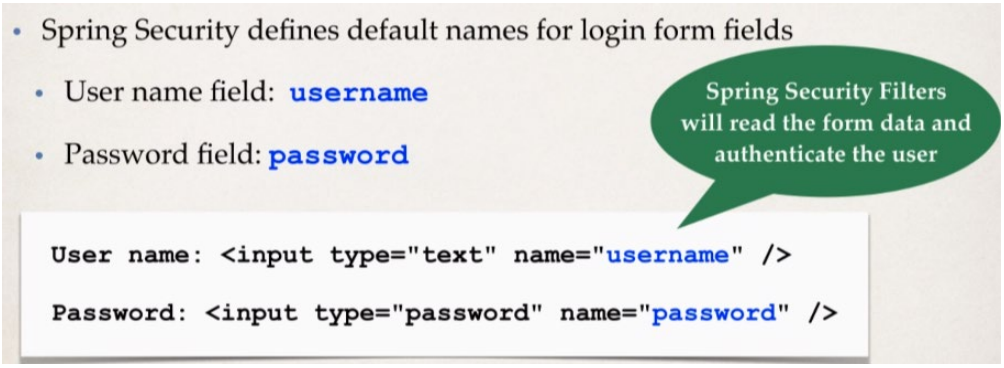


3. Create the actual login form using HTML (CSS optional)

we are going to send the login form data for processing to url: ***/authenticateTheUser***
This url is based on the configuration that we have setup in our config file.
We can give any values to the configuration url, we just need to be consistent in our app.



Login processing URL will be handled by Spring Security Filters.



Spring Security will read the data from the default names input field and authenticate the user.

Putting all together:

File: WEB-INF/view/plain-login.jsp

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
…
<form:form action="${pageContext.request.contextPath}/authenticateTheUser"
           method="POST">

  <p>
    User name: <input type="text" name="username" />
  </p>

  <p>
    Password: <input type="password" name="password" />
  </p>

  <input type="submit" value="Login" />

</form:form>
…
```

More on Context Path:

On the Eclipse world they call it context root and in JPS they call it context path.

So, this actually give us the access of the context path dynamically so that we can include it on the fly for any links we may have in our application for like form submissions, href links and so on.

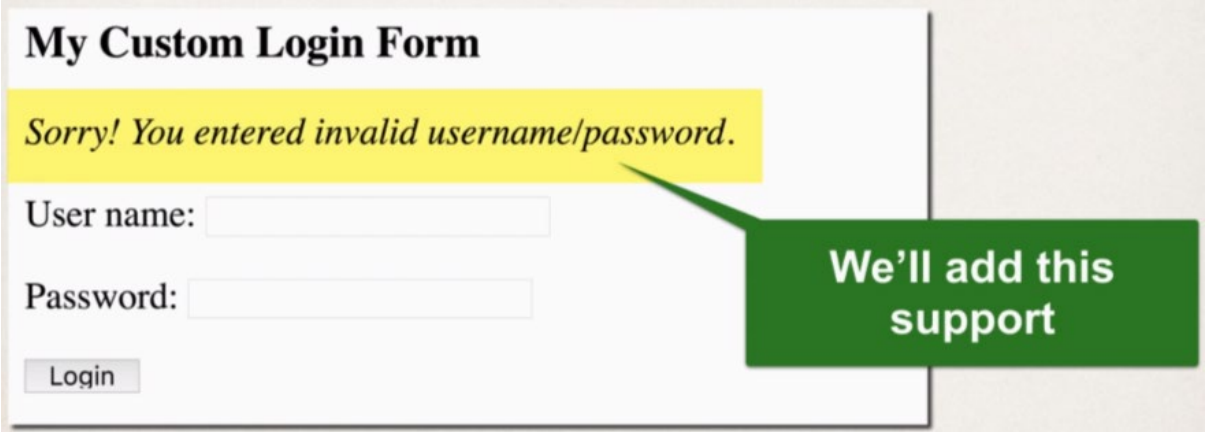
Why use Context Path?

- Allows us to dynamically reference context path of the application.
- Helps to keep links relative to application context path.
- So, if we change the context path of our application, then links will still work.
- And this is much better than hard coding the context path.

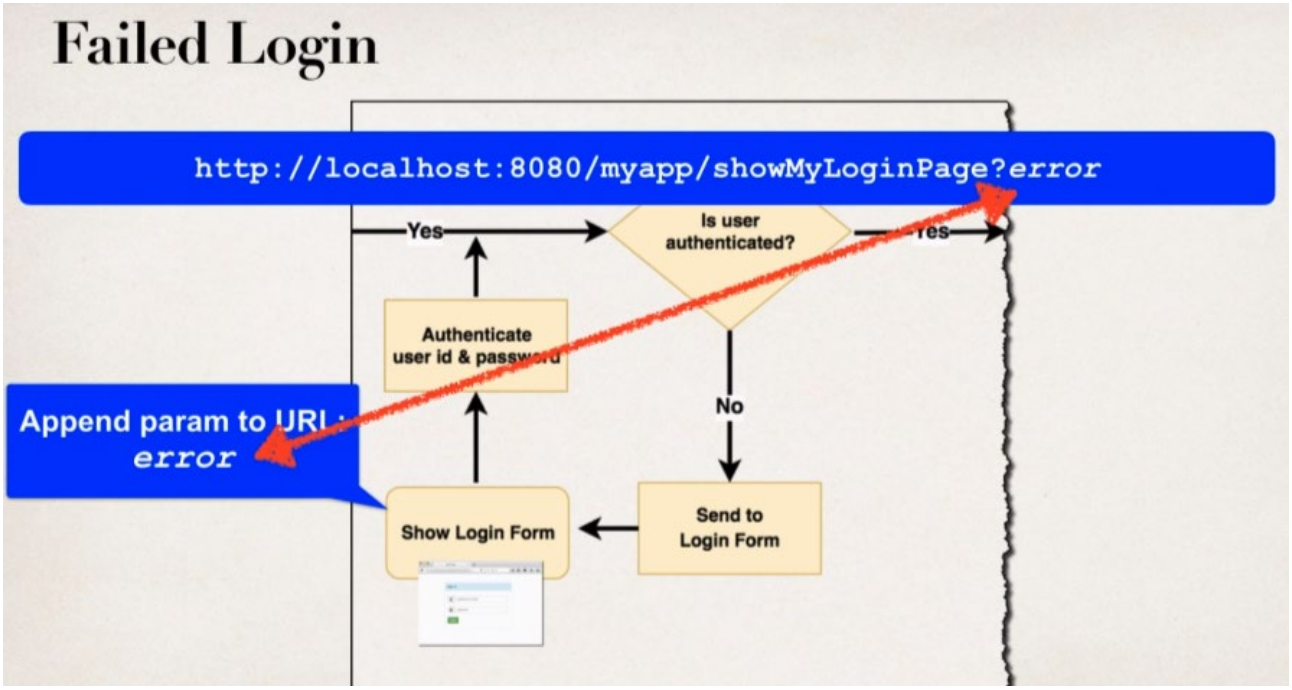
4. <u>And we can also make use of Spring MVC form tag</u>.

On entering the wrong credentials in our custom login page, there will be no error messages as we have seen in default login form because we need to do it on our own to handle the login error messages.

Adding Login Error message for our custom form:



When the user is getting authenticated and authentication gets failed then spring will send the user back to the login form and it will append the parameter **error** to the url and we can write code to check for that.



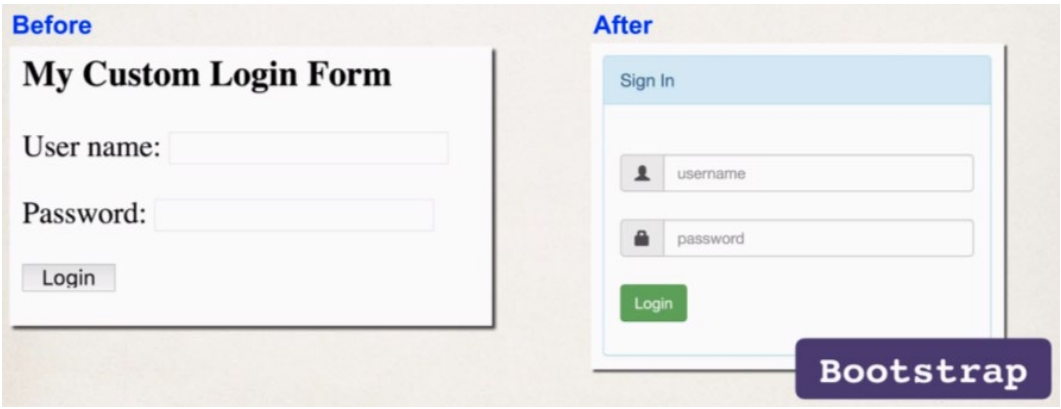
Development Process

1. Modify custom login form
2. Check the **error** parameter
3. If **error** parameter exists, show an error message

We can use JSTL to help with this.

Section 49: Spring Security - Bootstrap CSS Login Form

0 / 5 | 15min



Bootstrap:

- It is a web framework that includes CSS styles and Javascript.
- Focused on front-end UI

Development process:

1. Modify form and add support for Spring MVC form tags
2. Update form to point to our login processing URL
3. Verify form fields for username and password.
4. Change our controller to use our new Bootstrap login form.

FAQ: How To Add Local CSS file for Spring Security Login Form?

Question:
What would be required to link a CSS file for Spring Security login form? For example, I'd like to have a CSS file (from src/main/webapp/css/style.css) for the style of the error message when using Spring Security login form. I suspect that it block the request to the CSS if i am not authenticated.

Answer:
You are correct, by default Spring Security will block all requests to the web app if the user is not authenticated. However, if you'd like to use a local CSS file on your login form then these are the basic steps:

1. Create a separate CSS file

2. Reference the CSS file in your login page

3. Configure Spring Security to allow unauthenticated requests (permit all) to the "/css" directory

4. Configure the all Java configuration to serve content from the "/css" directory

Full Source code available here: spring-security-custom-login-form-with-css.zip

Here are the details of each step.

1. Create a separate CSS file
You can use any type of CSS styling. Just to demonstrate the point, the heading for the login page will be blue. The error messages will be red.

File: src/main/webapp/css/demo.css

```
.failed {
 color: red;
}

h3 {
 color: blue;
}
```

2. Reference the CSS file in your login page
In the head section of the login page, reference the CSS file. Here's the snippet

```
<head>
 <title>Custom Login Page</title>
 <link rel="stylesheet" type="text/css" href="css/demo.css">
</head>
```

and here's the complete code for this file.

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

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>

<head>
    <title>Custom Login Page</title>

    <link rel="stylesheet" type="text/css" href="css/demo.css">

</head>

<body>

<h3>My Custom Login Page</h3>

    <form:form action="${pageContext.request.contextPath}/authenticateTheUser"
               method="POST">

        <!-- Check for login error -->

        <c:if test="${param.error != null}">

            <i class="failed">Sorry! You entered invalid username/password.</i>

        </c:if>

        <p>
            User name: <input type="text" name="username" />
        </p>

        <p>
            Password: <input type="password" name="password" />
        </p>

        <input type="submit" value="Login" />

    </form:form>

</body>

</html>
```

3. Configure Spring Security to allow unauthenticated requests (permit all) to the "/css" directory

Make note of this snippet. It permits all requests to CSS. No need for authentication. Here's the snippet

```
        http.authorizeRequests()
                .antMatchers("/css/**").permitAll()
                .anyRequest().authenticated()
            .and()
            .formLogin()
                .loginPage("/showMyLoginPage")
                .loginProcessingUrl("/authenticateTheUser")
                .permitAll();
```

and here's the complete code for this file.

File: DemoSecurityConfig.java

```
1.  package com.luv2code.springsecurity.demo.config;
2.
3.  import org.springframework.context.annotation.Configuration;
4.  import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBu
    ilder;
5.  import org.springframework.security.config.annotation.web.builders.HttpSecurity;
6.  import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
7.  import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
8.  import org.springframework.security.core.userdetails.User;
9.  import org.springframework.security.core.userdetails.User.UserBuilder;
10.
11. @Configuration
12. @EnableWebSecurity
13. public class DemoSecurityConfig extends WebSecurityConfigurerAdapter {
14.
15.     @Override
16.     protected void configure(AuthenticationManagerBuilder auth) throws Exception {
17.
18.         // add our users for in memory authentication
19.
20.         UserBuilder users = User.withDefaultPasswordEncoder();
21.
22.         auth.inMemoryAuthentication()
23.             .withUser(users.username("john").password("test123").roles("EMPLOYEE"))
24.             .withUser(users.username("mary").password("test123").roles("MANAGER"))
25.             .withUser(users.username("susan").password("test123").roles("ADMIN"));
26.     }
27.
28.     @Override
29.     protected void configure(HttpSecurity http) throws Exception {
30.
31.         http.authorizeRequests()
32.                 .antMatchers("/css/**").permitAll()
33.                 .anyRequest().authenticated()
34.             .and()
35.             .formLogin()
36.                 .loginPage("/showMyLoginPage")
37.                 .loginProcessingUrl("/authenticateTheUser")
38.                 .permitAll();
39.     }
40.
41. }
```

4. Configure the all Java configuration to serve content from the "/css" directory

Update your `DemoAppConfig` to implement the `WebMvcConfigurer` interface. This allows you to override a method for serving up content from the "/css" directory. Here's the snippet

```
1.   @Override
2.   public void addResourceHandlers(ResourceHandlerRegistry registry) {
3.       registry.addResourceHandler("/css/**").addResourceLocations("/css/");
4.   }
```

and here's the complete code for this file.

```
1.  package com.luv2code.springsecurity.demo.config;
2.
3.  import org.springframework.context.annotation.Bean;
4.  import org.springframework.context.annotation.ComponentScan;
5.  import org.springframework.context.annotation.Configuration;
6.  import org.springframework.web.servlet.ViewResolver;
```

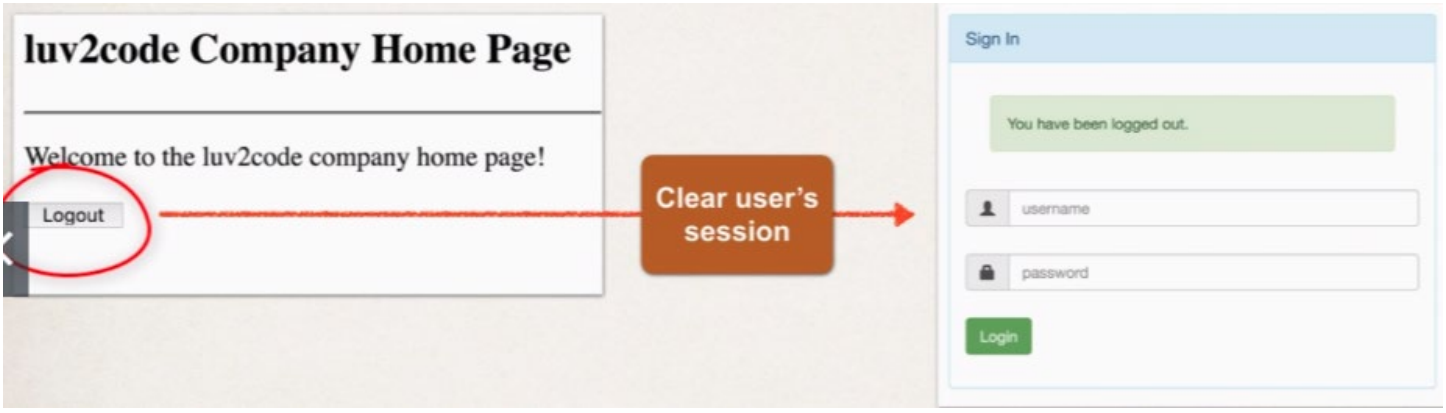
```
7.  import org.springframework.web.servlet.config.annotation.EnableWebMvc;
8.  import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
9.  import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
10. import org.springframework.web.servlet.view.InternalResourceViewResolver;
11.
12. @Configuration
13. @EnableWebMvc
14. @ComponentScan(basePackages = "com.luv2code.springsecurity.demo")
15. public class DemoAppConfig implements WebMvcConfigurer {
16.
17.     // define a bean for ViewResolver
18.
19.     @Bean
20.     public ViewResolver viewResolver() {
21.
22.         InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
23.
24.         viewResolver.setPrefix("/WEB-INF/view/");
25.         viewResolver.setSuffix(".jsp");
26.
27.         return viewResolver;
28.     }
29.
30.     @Override
31.     public void addResourceHandlers(ResourceHandlerRegistry registry) {
32.         registry.addResourceHandler("/css/**").addResourceLocations("/css/");
33.     }
34. }
```

Once you apply these changes then you'll see use of the CSS on the login form. When login form is first loaded, you'll see the blue heading based on CSS style. If there is a failed login, you'll see the red error message based on CSS style.

Section 50: Spring Security - Adding Logout Support
0 / 3 | 11min

Logging out

We will add the logout button our home page which will clear the user's session and redirect the user to the login page.



Development Process:

1. Add logout support to spring security configuration.

```
@Override
protected void configure(HttpSecurity http) throws Exception {

  http.authorizeRequests()
        .anyRequest().authenticated()
      .and()
      .formLogin()
        .loginPage("/showMyLoginPage")
        .loginProcessingUrl("/authenticateTheUser")
        .permitAll()
      .and()
      .logout().permitAll();
}
```

Add logout support for default URL /logout

So, the last two lines will basically give logout support that will expose the default URL (/logout) for logging out.

2. Add logout button to JSP page

- Send data to default logout URL: `/logout`
- Logout URL will be handled by Spring Security Filters.

```
<form:form action="${pageContext.request.contextPath}/logout"
           method="POST">

      <input type="submit" value="Logout" />

</form:form>
```

We need to POST data to URL `/logout`.

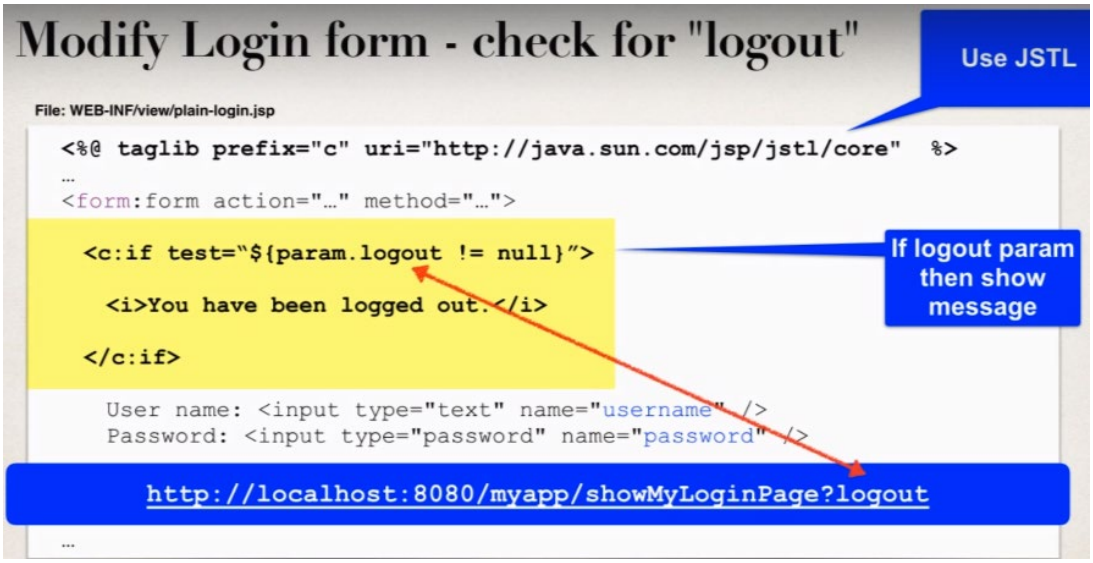
When a logout is processed, by default Spring Security will

- Invalidate user's HTTP session and remove session cookies, etc.
- Send user back to our login page
- Append a logout parameter: ***?logout***

3. Update login form to display "Logged out" message.

- We will check for the logout parameter
- If logout parameter exists, show "logged out" message.

And again we can make use of JSTL for logout.



Section 51: Spring Security - Cross Site Request Forgery (CSRF)

0 / 3 | 14min

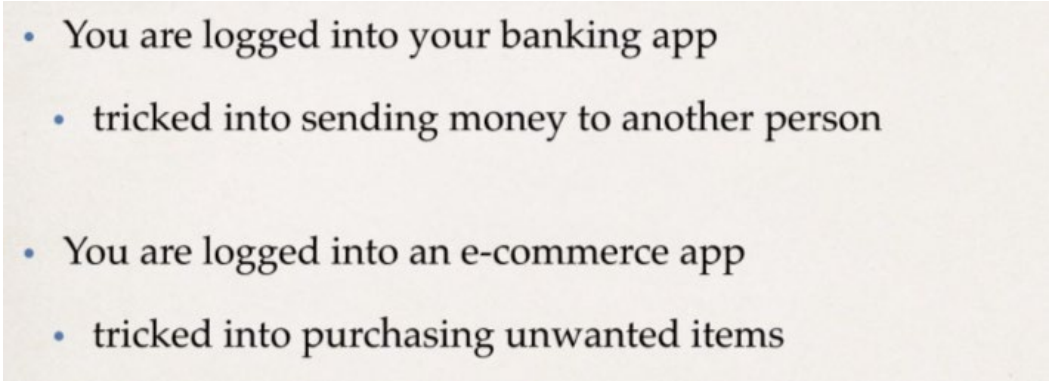
Cross Site Request Forgery (CSRF):

As we know <form:form> tag provides automatic support for security defenses.

Spring Security protects against Cross Site Request Forgery.

What is CSRF?
A security attack where an evil website tricks you into executing an action on a web application that you are currently logged in.

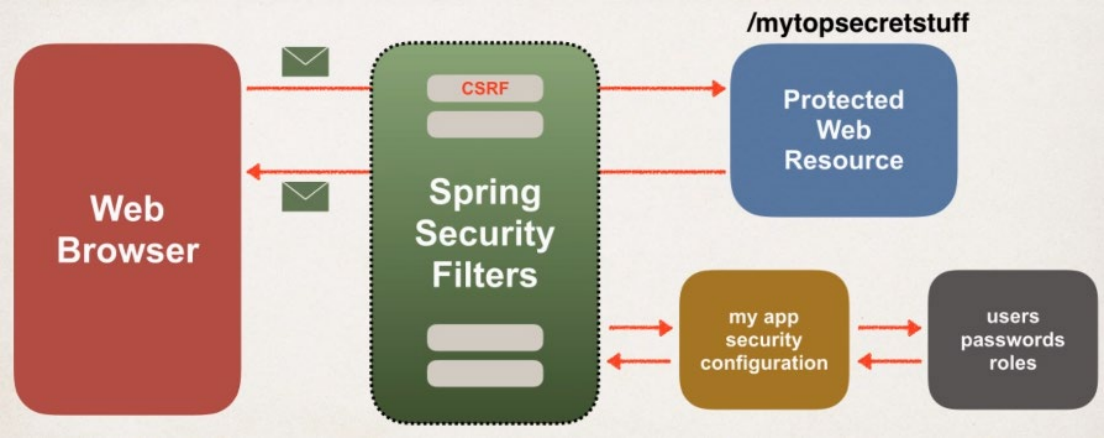
CSRF Examples:



CSRF Protection:

- To protect against CSRF attacks
- Embed additional authentication data / token into all HTML forms
- On subsequent requests, web app will verify token before processing.

Spring Security Filters can generate tokens to send back to the browser, we can use it in our HTML forms and can send the data accordingly. And on any incoming requests spring security filters will verify the tokens to make sure that they are valid for this given user session before even processing the request.



- CSRF protection is enabled by default in spring security
- Spring Security uses the synchronizer token pattern
  - Where each request includes a session cookie and randomly generated token
- And for the request processing, spring security verifies token before processing.
- And all of this is handled behind the scenes by spring security filters.

When to use CSRF Protection?

The Spring security team recommends to use the CSRF protection for any normal browser web requests.

If you are building a service for now browser client then you may want to disable CSRF protection.

Manually add CSRF token

```
<form action="..." method="POST">

        <input type="hidden"
               name="${_csrf.parameterName}"
              value="${_csrf.token}" />

</form>
```

Manually add CSRF token

What happens if you don't include CSRF token?

Let's Break It!

HTTP Status 403 – Forbidden

Type Status Report

Message Invalid CSRF Token 'null' was found on the request parameter '_csrf' or header 'X-CSRF-TOKEN'.

Description The server understood the request but refuses to authorize it.

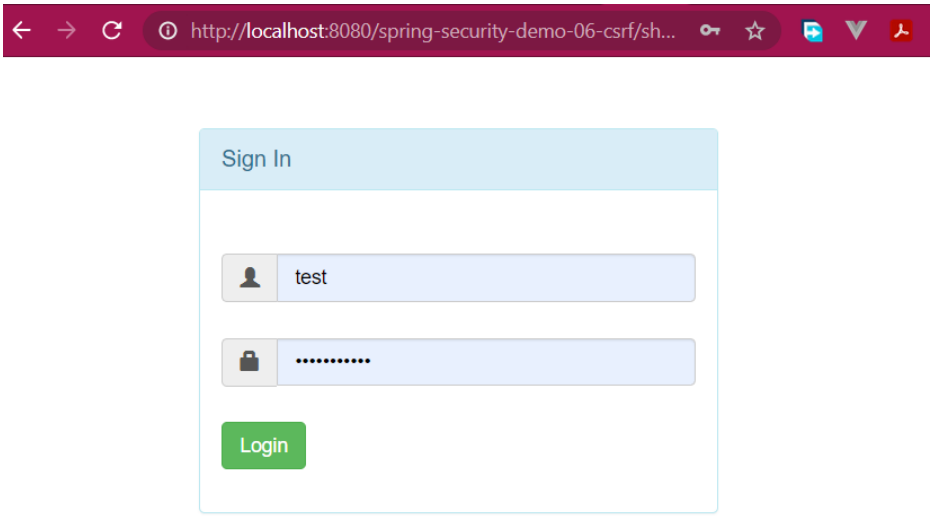
Token was not included

CSRF Resources

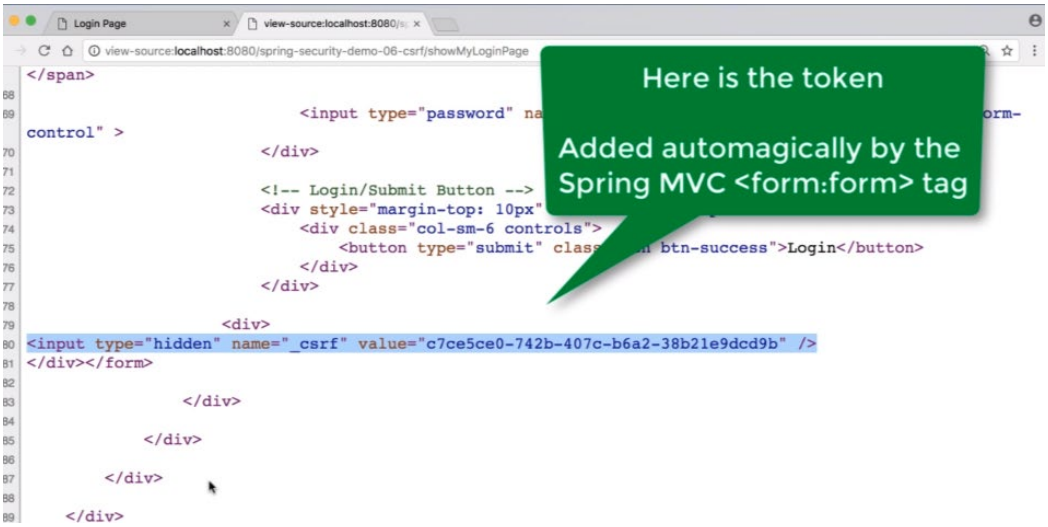
- CSRF Security Reference
  - **https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)**
- Spring Security CSRF Support
  - **https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#csrf**

Viewing CSRF Tokens

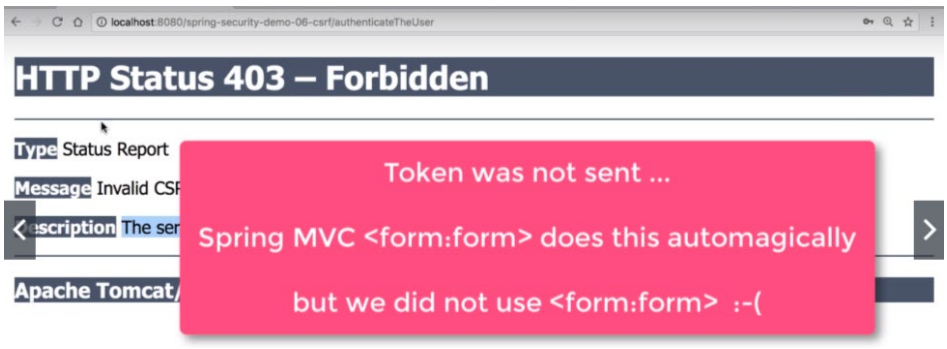
Open the app in the browser and do **right click** and then click on **view page source.**



And then we will get to see the csrf token inserted automatically by the <form:form> tag, which spring security filters will get this token along with a session cookie and then spring MVC will compare whether it works for the current user session before processing the request.



And in case we remove the `form:form` tag and use normal `form` tag then we will get error as we are not providing the csrf token with the requests.



Manually adding CSRF token:

```
<!-- User name -->
<div style="margin-bottom: 25px" class="input-group">
    <span class="input-group-addon"><i class="glyphicon glyphi

    <input type="text" name="username" placeholder="username"
</div>

<!-- Password -->
<div style="margin-bottom: 25px" class="input-group">
    <span class="input-group-addon"><i class="glyphicon glyphi

    <input type="password" name="password" placeholder="passwo
</div>

<!-- Login/Submit Button -->
<div style="margin-top: 10px" class="form-group">
    <div class="col-sm-6 controls">
        <button type="submit" class="btn btn-success">Login</b
    </div>
</div>

<!-- I am manually adding the tokens... -->
<input  type="hidden"
name="${_csrf.parameterName}"
value="${_csrf.token}"/>
</form>
```

On getting the **page source**:

So, a randomly generated token generated for this given user session. Spring security will add the appropriate value and name to the csrf token.

```
    <span class="input-group-addon"><i class="glyphicon glyphicon-lock"></i><

    <input type="password" name="password" placeholder="password" class="form
</div>

<!-- Login/Submit Button -->
<div style="margin-top: 10px" class="form-group">
    <div class="col-sm-6 controls">
        <button type="submit" class="btn btn-success">Login</button>
    </div>
</div>

<!-- I am manually adding the tokens... -->
<input  type="hidden"
name="_csrf"
value="85e7c0d1-e746-48c8-ad12-8d541b261c19"/>
</form>

</div>
```

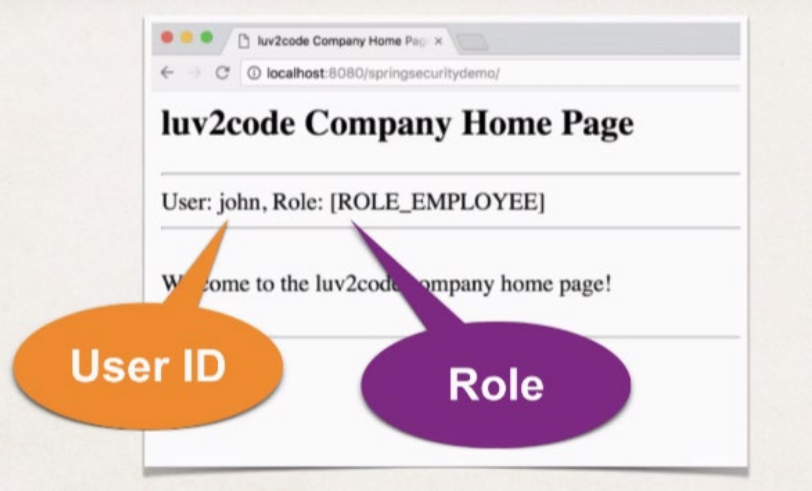
Section 52: Spring Security - User Roles

0 / 3 | 12min

Displaying the user Id and Roles:

Spring security provides JPS custom tags for accessing the user id and roles.

And it makes easy for us to add the support onto the JSP page.



Development Process

- Update POM file for spring security JSP tag library.
- Add Spring security JSP tag library to JSP page.
- Display the **userId** and **user roles**.

Step 3: Display User ID

File: home.jsp

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

User: <security:authentication property="principal.username" />
```

User ID

Step 4: Display User Roles

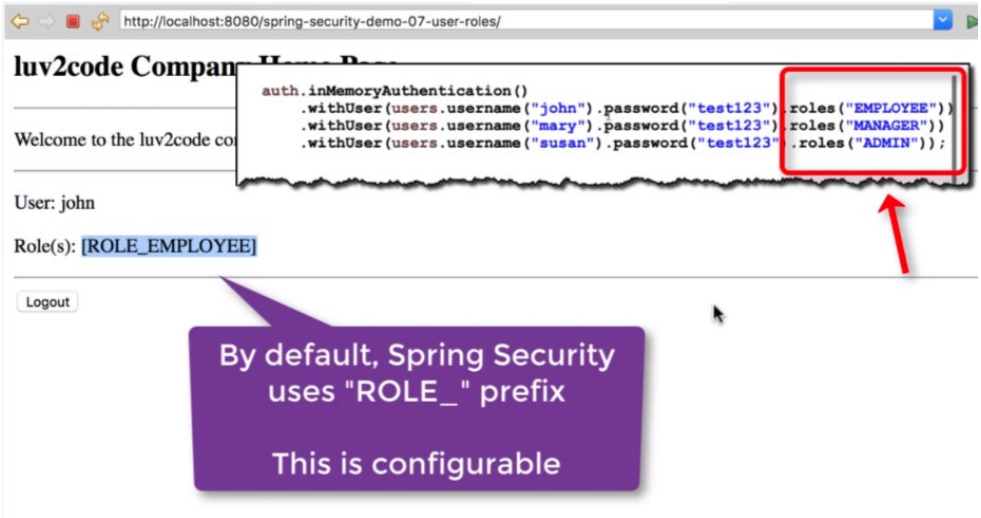
File: home.jsp

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

Role(s): <security:authentication property="principal.authorities" />
```

authorities is same as user roles

User Roles

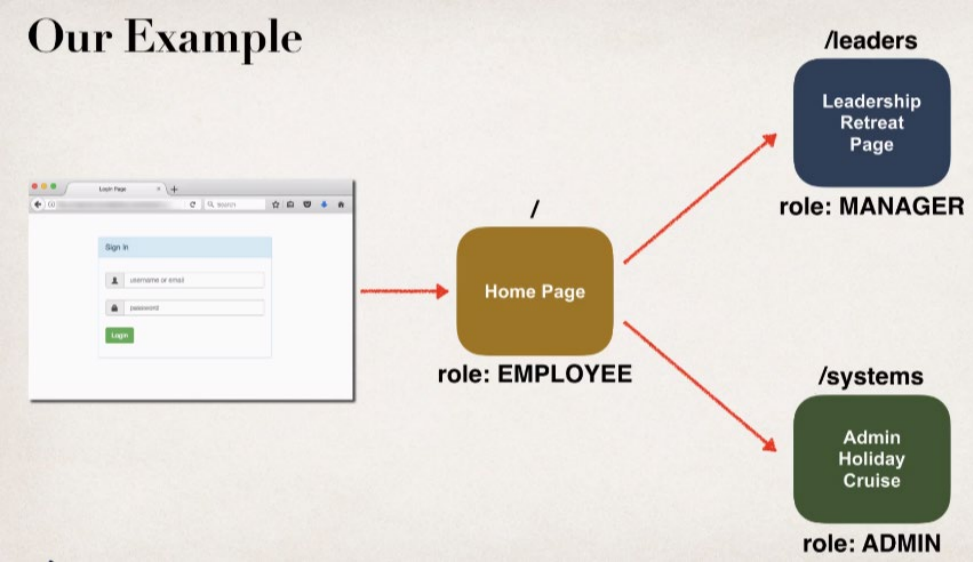
On output:



Section 53: Spring Security - Restrict Access Based on Role

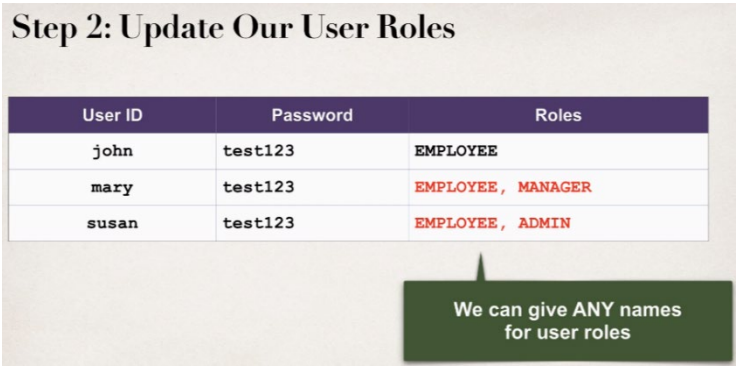
0 / 11 | 48min

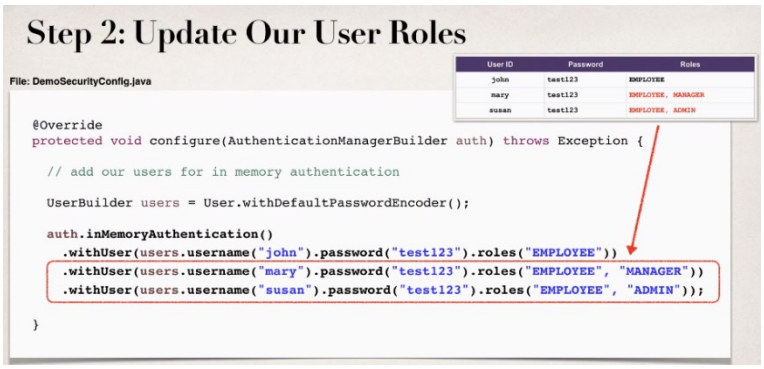
- **Restrict Access Based on Roles**



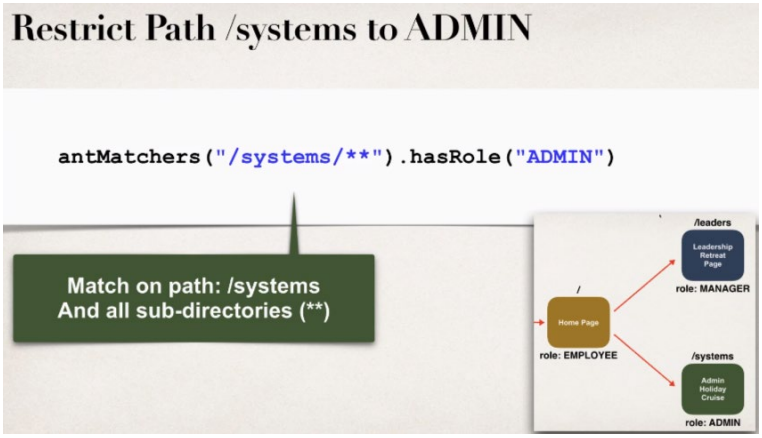
Development Process

1. Create supporting controller code and view pages.
2. Update user roles.



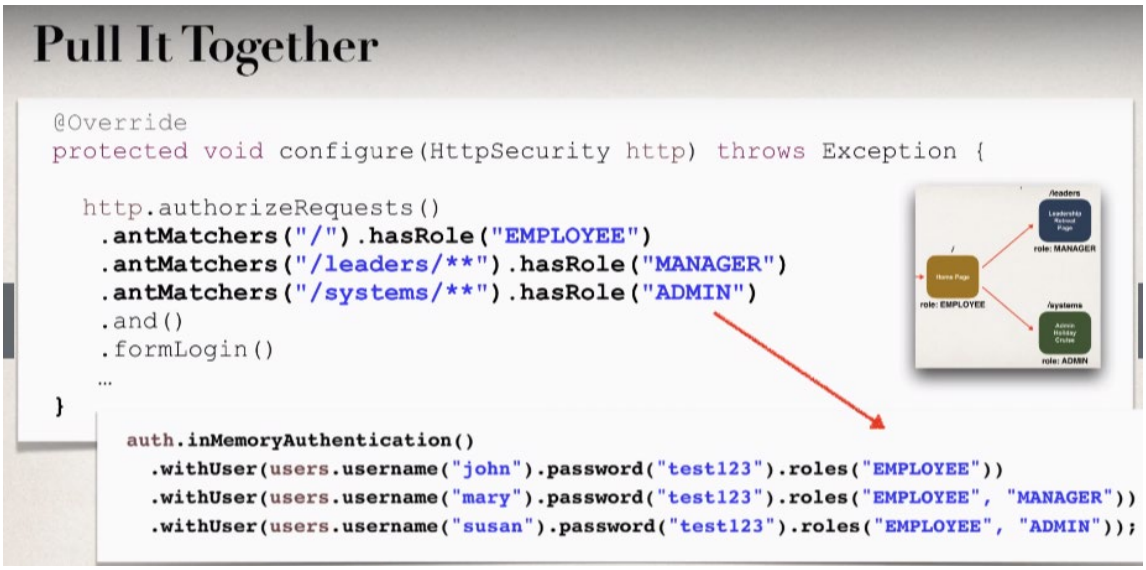


3. Restrict Access based on Roles.

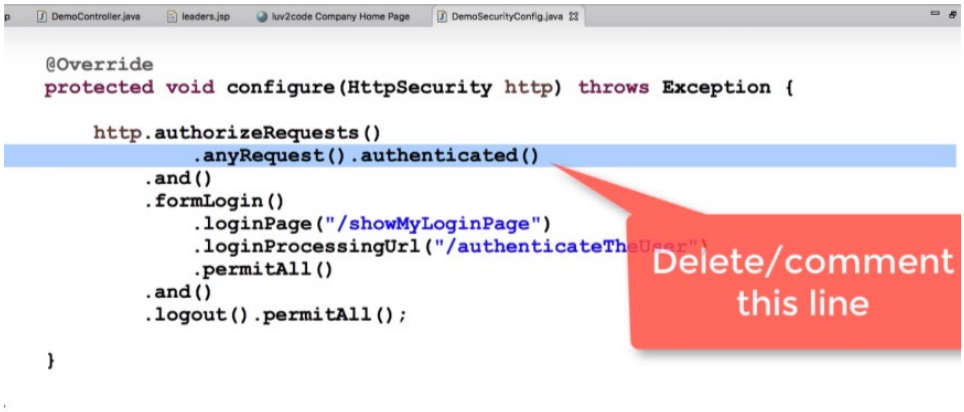


Anyone can access the home page if it has role “EMPLOYEE”, then the below code can be used to restrict the access:

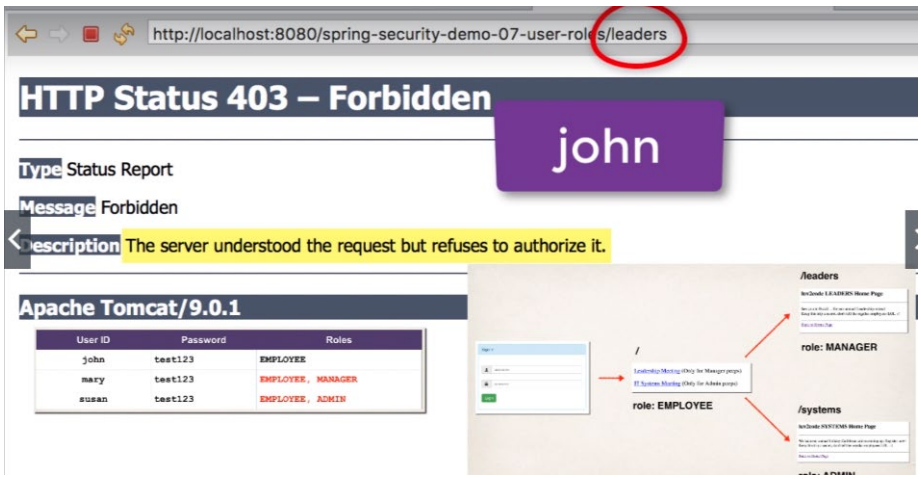
```
                    .antMatchers("/").hasRole("EMPLOYEE")
```



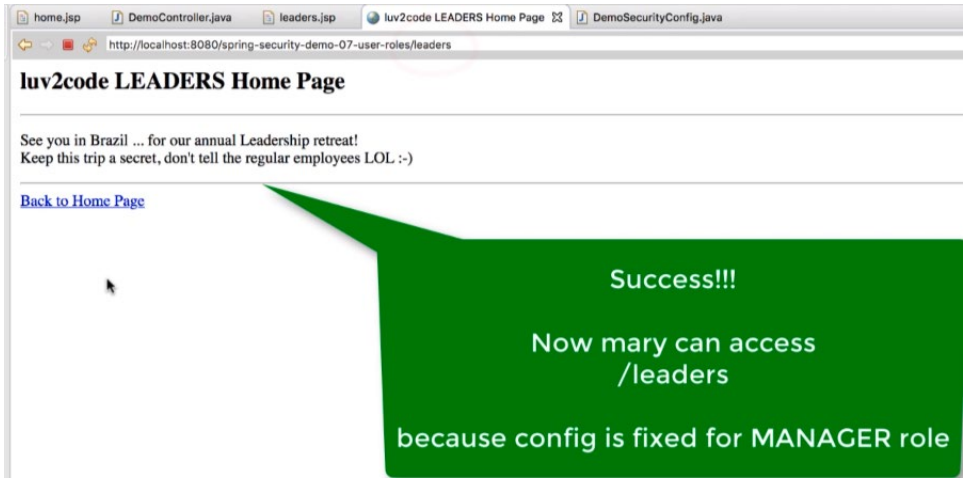
We will delete the highlighted line from our `demoConfig` class because we want to restrict the access based on specific roles not just user being authenticated.



Now if user having role “EMPLOYEE” want to access the /leaders url resources then spring security will throw error

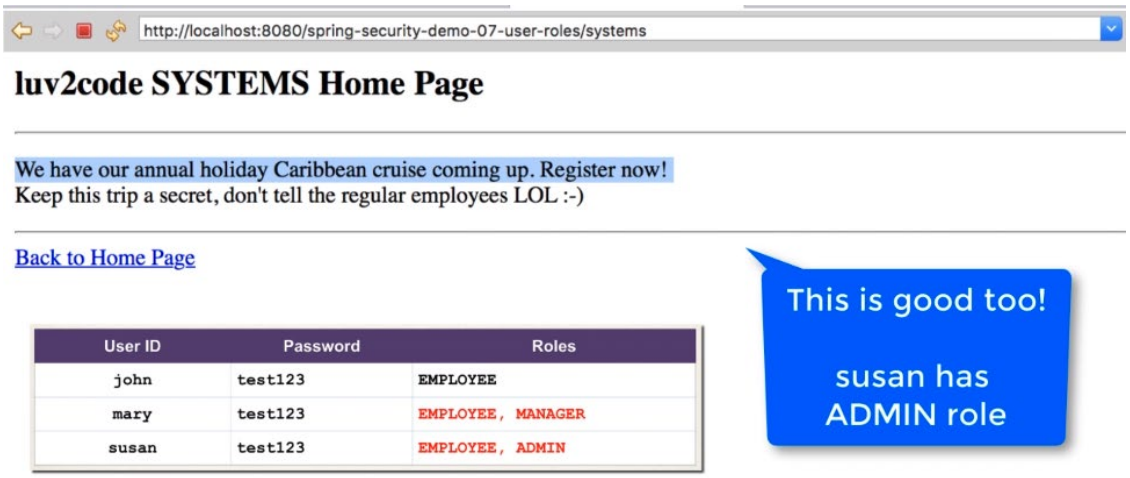


If **mary** who is having role **MAMAGER** as well as **EMPLOYEE** then she should be able to access the **/leaders** url.



Now we will test for ADMIN roles:

Since, in our example “susan” having admin role and she will be able to get through **/admins** url.



So, we are able restrict access based on roles.

➢ Create a Custom “Access Denied” Page:

So, we have seen the custom access denied page and it’s pretty scary.

Therefore, we will provide our own custom access-denied page with our own look and feel.

Development Process:

1. Configure custom page for access denied.

Here we have *exceptionalHandling().accessDeniedPage("/access-denied");*
This request mapping path will be used if there is an authorization error or user can not access a given page then spring security will use this request mapping path to show the user access-denied page.

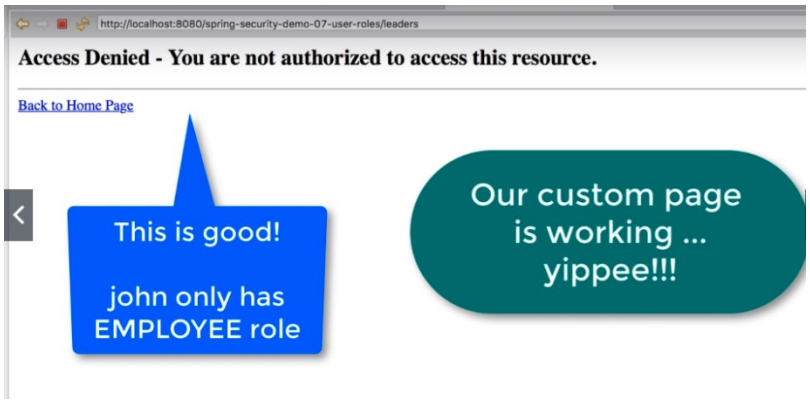
2. Create supporting controller code and view page.

File: LoginController.java

```
//add request mapping for /access-denied
@GetMapping("/access-denied")
public String showAccessDeniedPage() {
	return "access-denied";
}
```

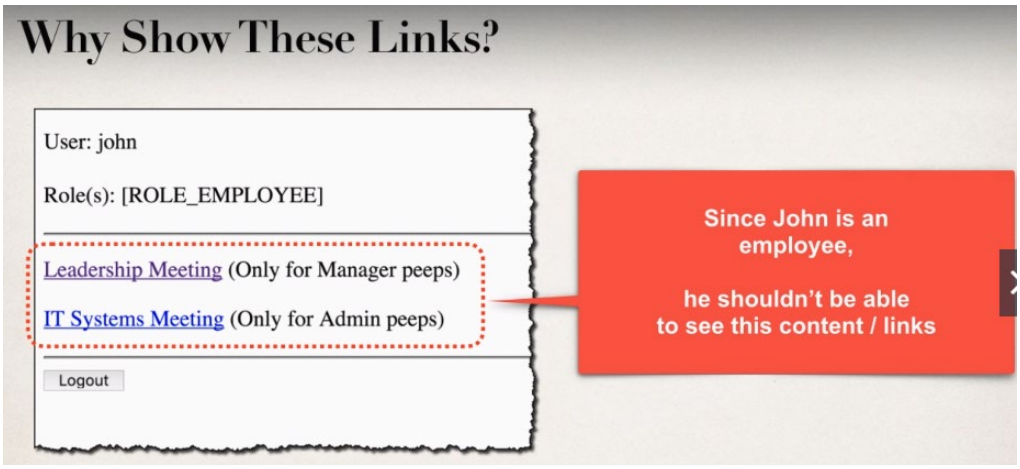
File: access-denied.jsp

```
<body>
	<h2>Access Denied - you are not authorized to access this resource</h2>
	<a href="${pageContext.request.contextPath}">Back to Home Page</a>
</body>
```

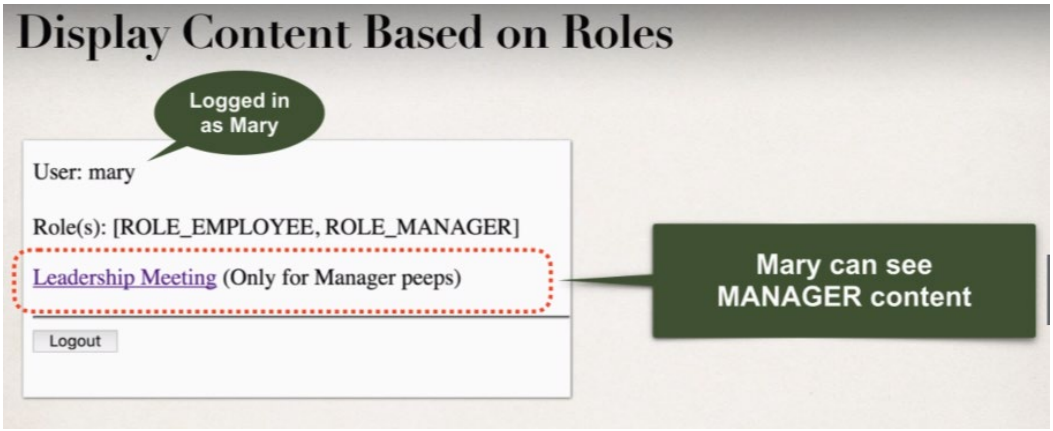


On check with **john** user since he has only EMPLOYEE role, he can't access the Managerial page and then our custom built access-denied page will pop up for him.

➢ **Display Content Based on Roles:**



I would like to display content on the basis of role. So, **we can make use of spring security JSP tag to display content based on user's Role.**

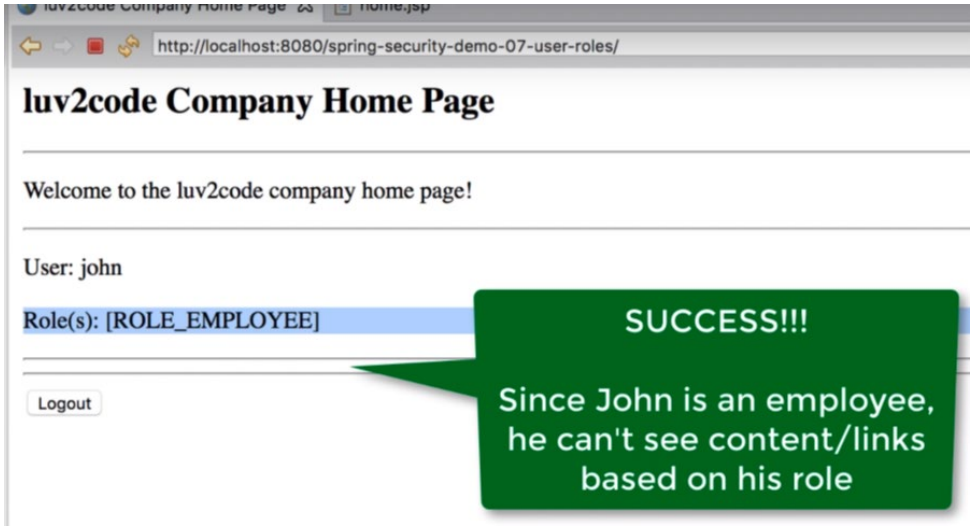


Here we are displaying info based on **Role = Manager**.

- Even if there is some user who knows about web browser the user may attempt to view document source then content encircled with red color will not be hidden, it will simply be not included i.e., the resultant html page will not even have this content.

Output:

Since, John is an employee therefore he can't see any of the links that doesn't belong to his role.



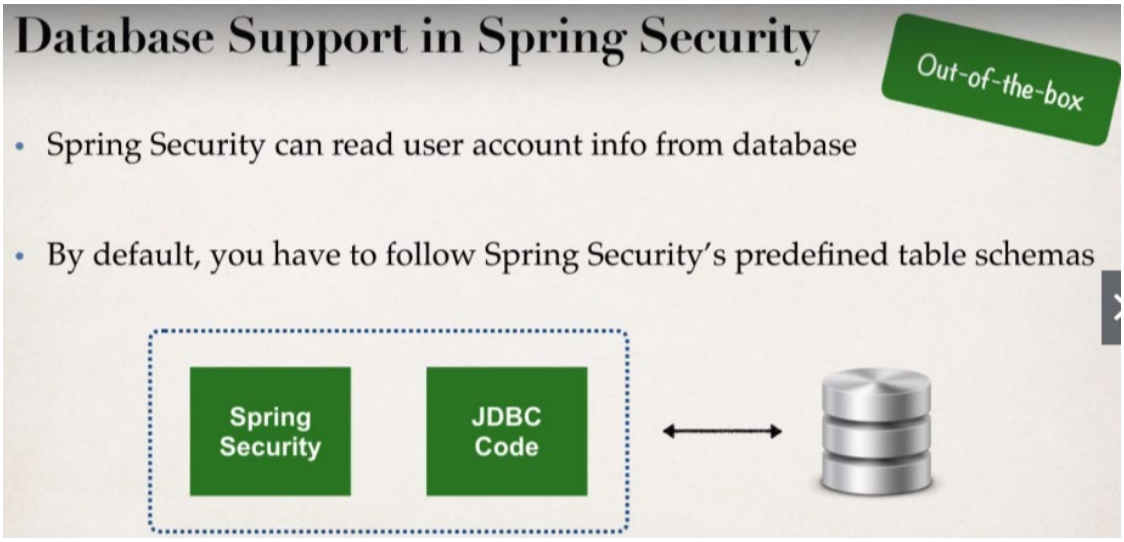
Section 54: Spring Security - Add JDBC Database Authentication

0 / 11 | 52min

- **User Accounts Stored in Database:**

So far, our user accounts were hard coded in Java source code.

We want to add database access.



Spring Security includes all of the JDBC code to read information from the database. So, there is very little java code that we have to write as far as JDBC code for reading from the database.

We can also Customize the table schemas.

- Useful if you have custom tables specific to your project / custom.
- And then you will be responsible for developing the code to access the data
- Then we have to write low level JDBC, Hibernate Code to read from the appropriate tables.

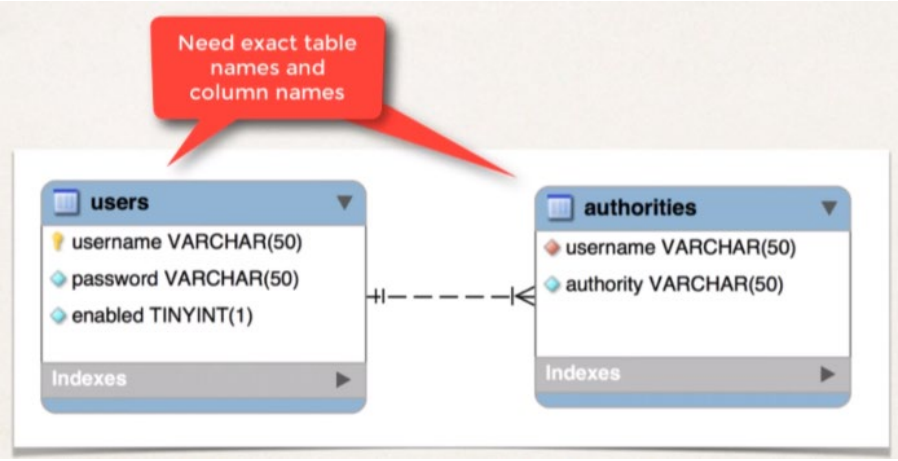
In our Example

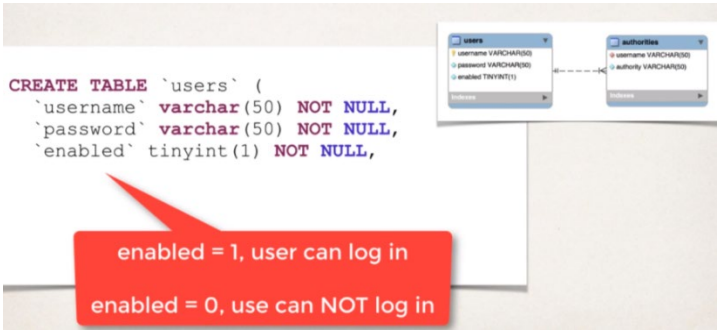
We will follow Spring Security's predefined table schemas.

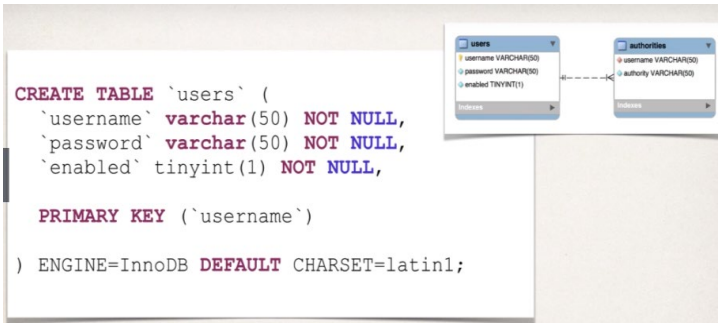
Development Process:

1. Develop SQL Script to set up database tables.

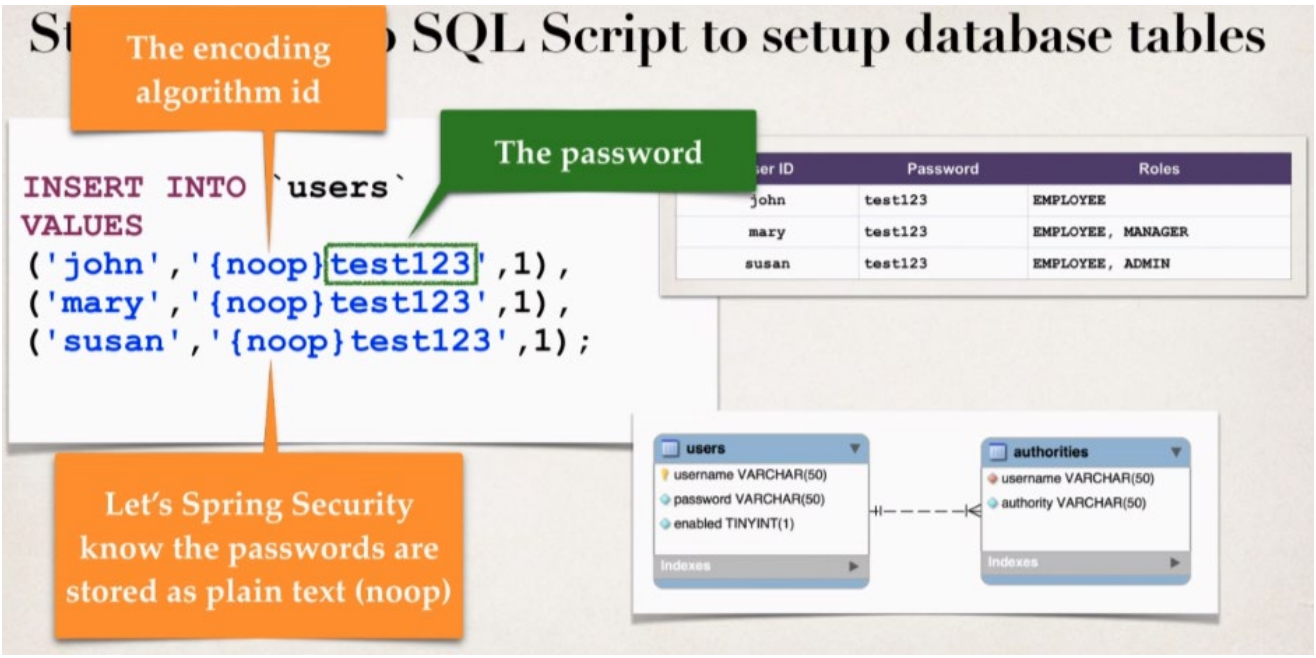
   We need to provide the same table name and columns in the schema definition and "authorities" is same as "roles".



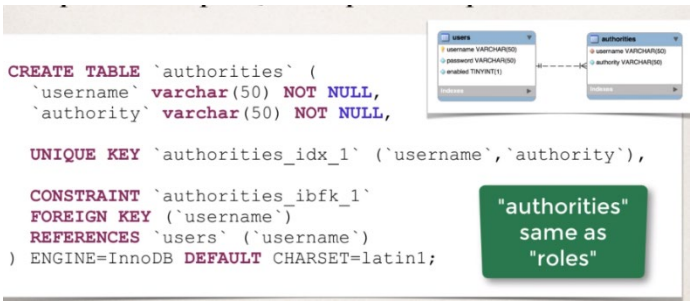


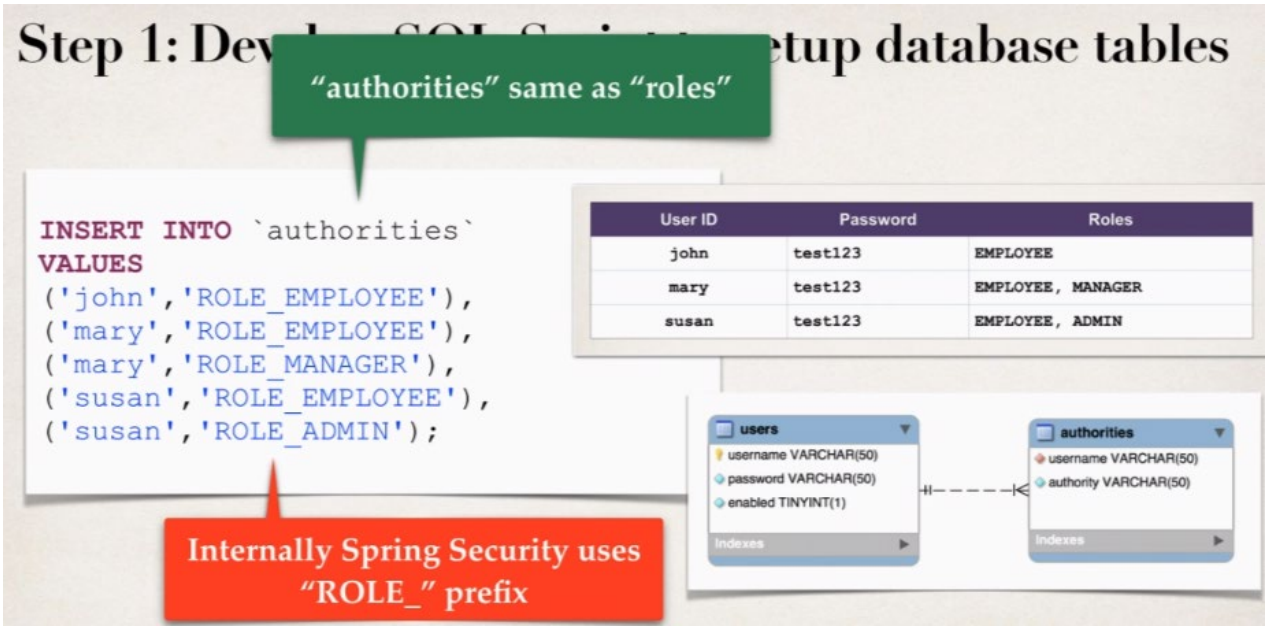


- **Id** is the encoding algorithm that is being used for this password.
- **Bcrypt** which is one way hashing technique, it's very popular and highly recommended by Spring Security.



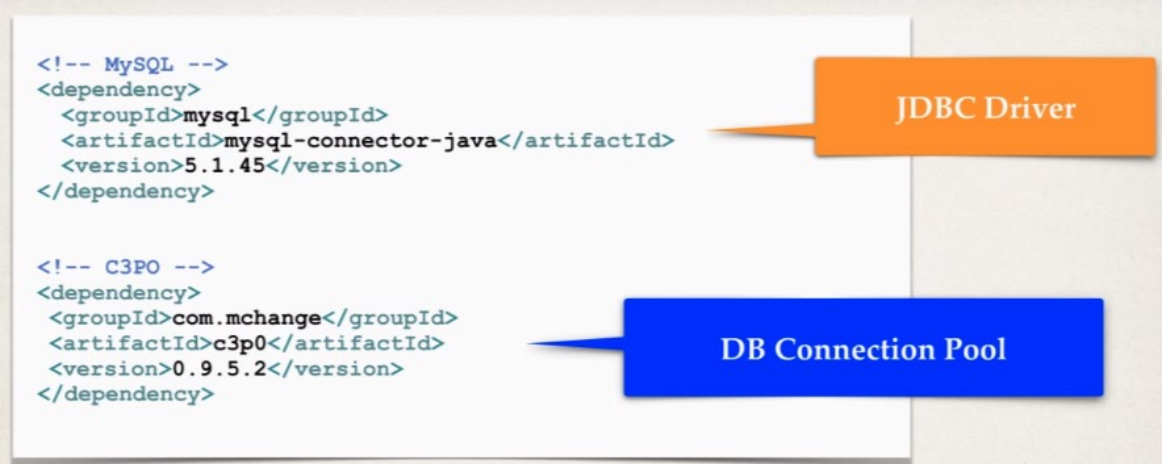
Now here we are creating the **authorities** schema as well:





2. <u>Add Database support to Maven POM File.</u>

   We need to give the reference of the MySQL JDBC Driver, as well as Connection Pooling so we will also give reference of **c3P0** and will give appropriate maven coordinates for that.



3. <u>Create JDBC properties file</u>

   We'll read this property file in our Java configuration class.

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

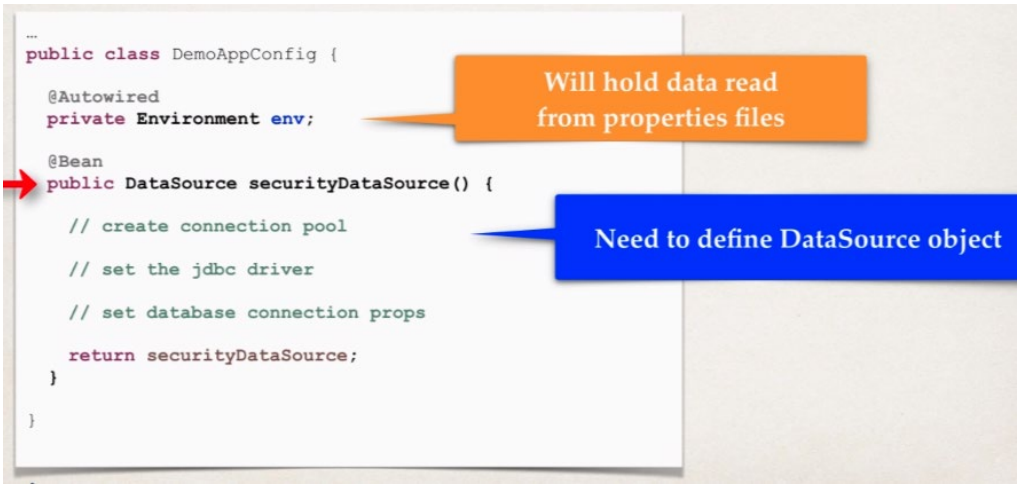
```
#
# JDBC connection properties
#
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/spring_security_demo?useSSL=false
jdbc.user=springstudent
jdbc.password=springstudent

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

4. Define DataSource in spring Configuration.

   In our configuration class, we will make use of ***@PropertySource("classpathh:persistence-mysql.properties")*** which contains the reference for this property file.

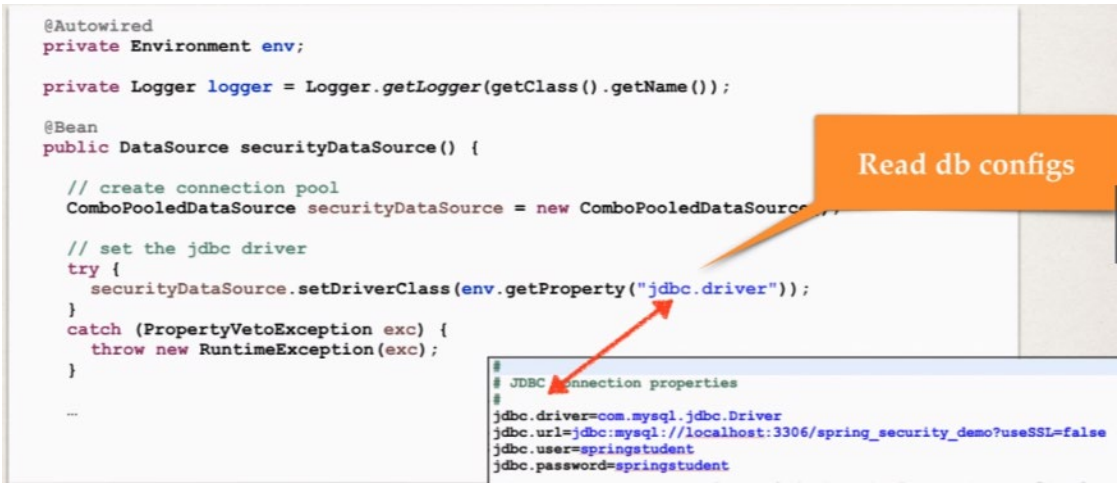
During the Maven build this properties file will automatically copied to classpath, since it's a web app, the file will get copied to the `WEB-INF` classes directory and it will on to the classpath and then we can read this property file from the classpath.

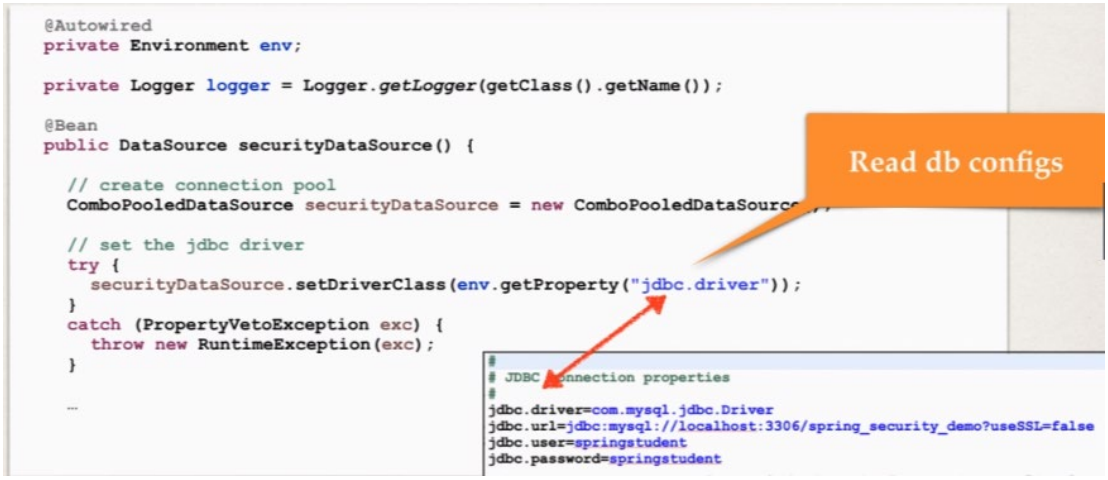
This is all handled by pure Java configuration.

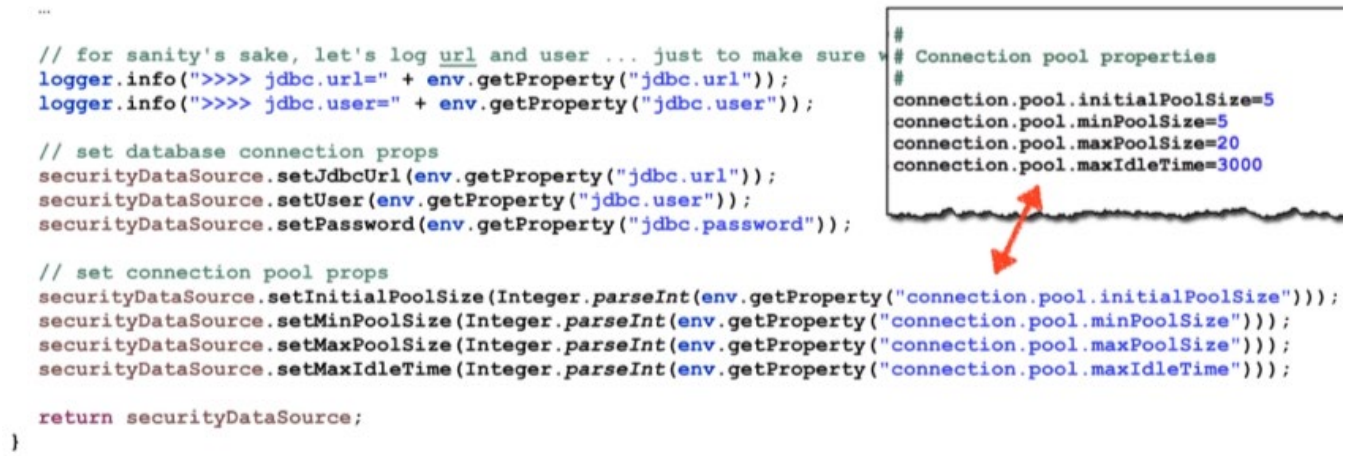


`Environment` is a spring helper class which is going to hold the data that has to be read from the properties file and we have already done the code for loading from the property file that data will be injected into this Environment object so that we can make use of it.

In the code below, we are setting up the JDBC Driver on our connection pool.





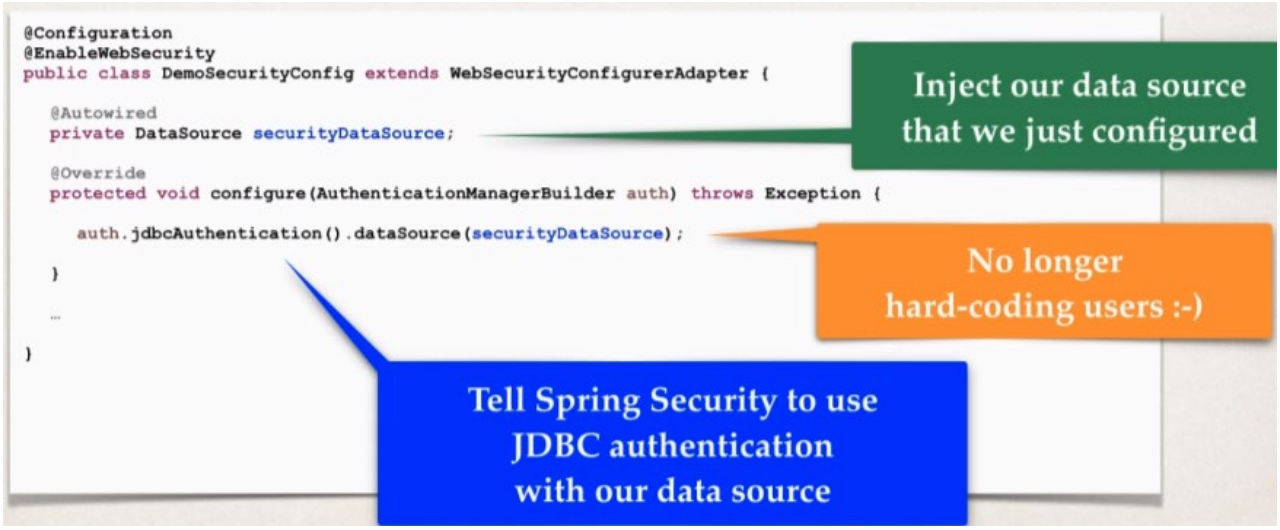


Now our bean will get created in within the **`securityDataSource`** method and we will do the Autowired into our **`DemoSecurityConfig`** class.

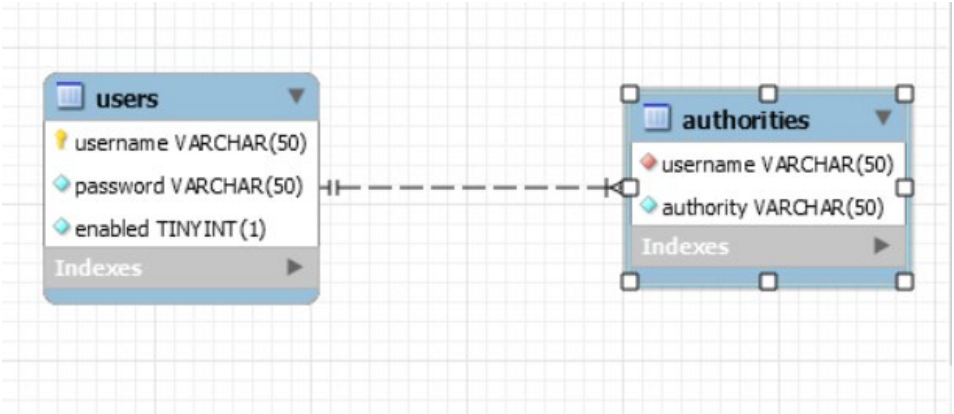
5. Update Spring Security Configuration to use JDBC.

So, in our DemoSecurityConfig class we are injecting the Datasource that we have configured above. Now when I am overriding the configure method with AuthenticateManagerBuilder argument, we are telling the Spring security to use JDBC Authentication and we are providing the injected data source.

And we are no longer hard coding the users and we are reading them from the DB. Now, spring security knows about our DB as we have setup all the things into our **`SecurityConfiguration`**, and spring security will handle all the low-level work of reading the user, password and roles and so on.



- Here is the link to the DB **SQL SCRIPT**.



This is basically predefined table schema.

- Now we need to put the properties file in the **`src>main>resources`** folder.
  Because it's the standard maven directory and it's recognized by maven and it will be used in build process when we build and run our application.

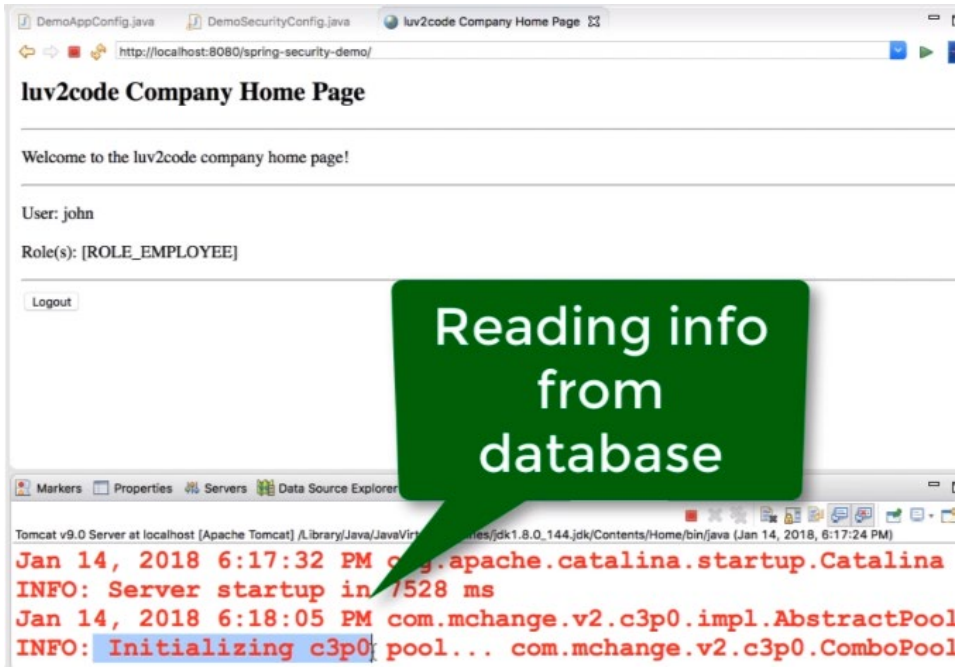
  Properties file link

  And whenever we read form the property file value will always come as a **string**.

spring-security-demo-08-jdbc-plaintext/pom.xml | persistence-mysql.properties

```
#
# JDBC connection properties
#
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/spring_security_demo_plaintext?useSSL=false&serv
jdbc.user=springstudent
jdbc.password=springstudent

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

On Running the application on Server:



Section 55: Spring Security - Password Encryption

1 / 9 | 23min

Password Storage:

So far, our user passwords are stored in plaintext.

But it's not ready for production.

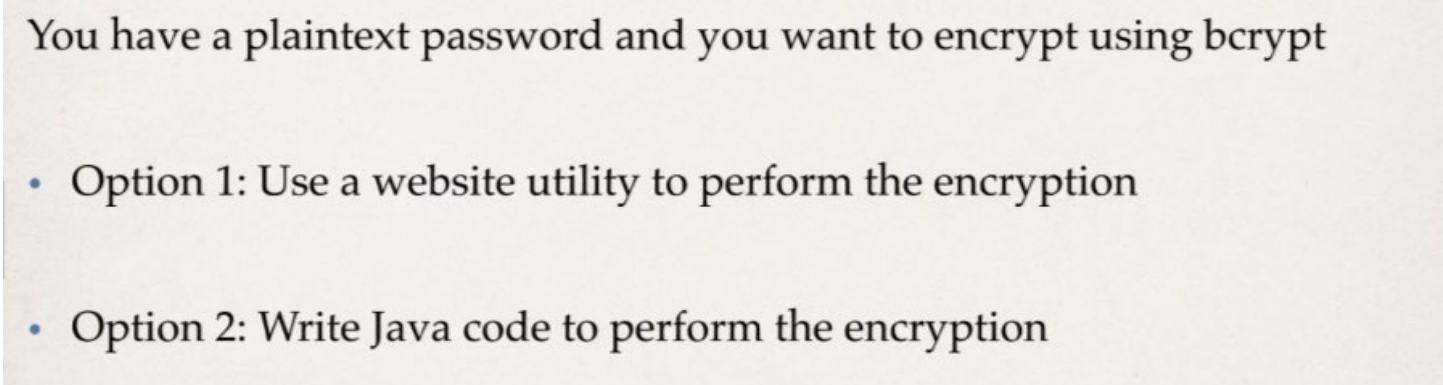
The Best practice is store passwords in an encrypted format.

Spring security recommends using the popular **bcyrpt** algorithm.

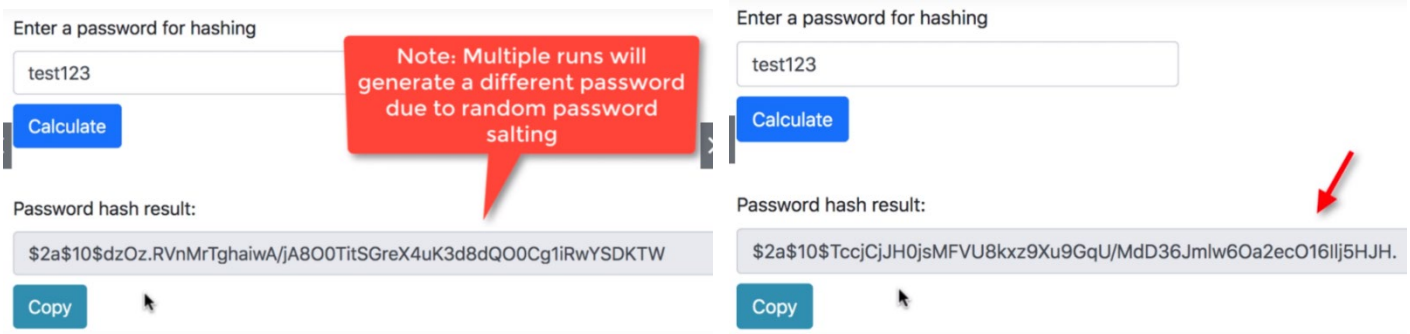
Bcyrpt Algo:

- Performs one-way encrypted hashing.
- Add a random salt to the password for additional protection
- Also includes support to defeat brute force attacks.

How to get a Bcrypt password:



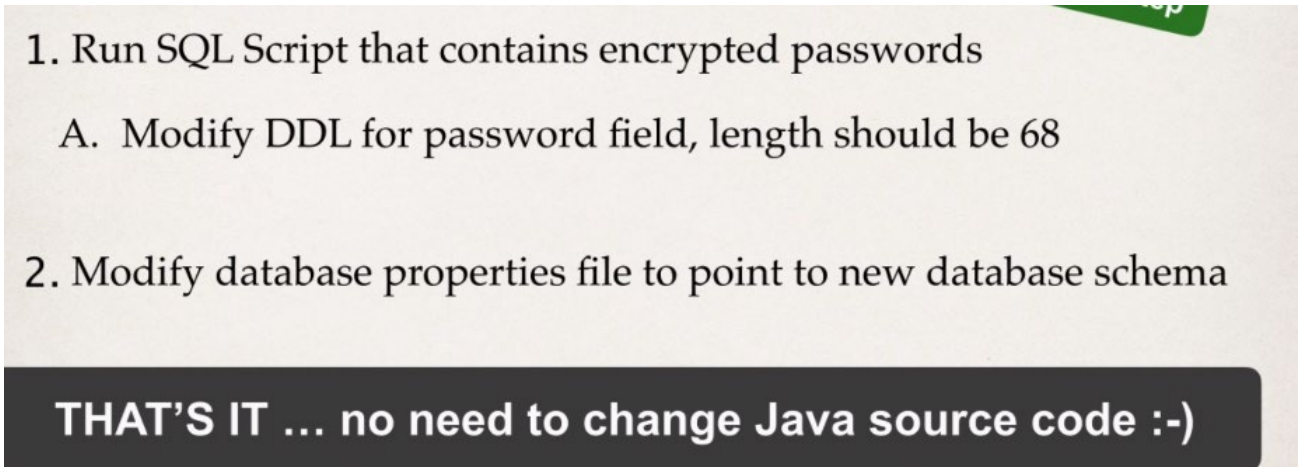
Here we will be using option 1: visit https://www.bcryptcalculator.com/

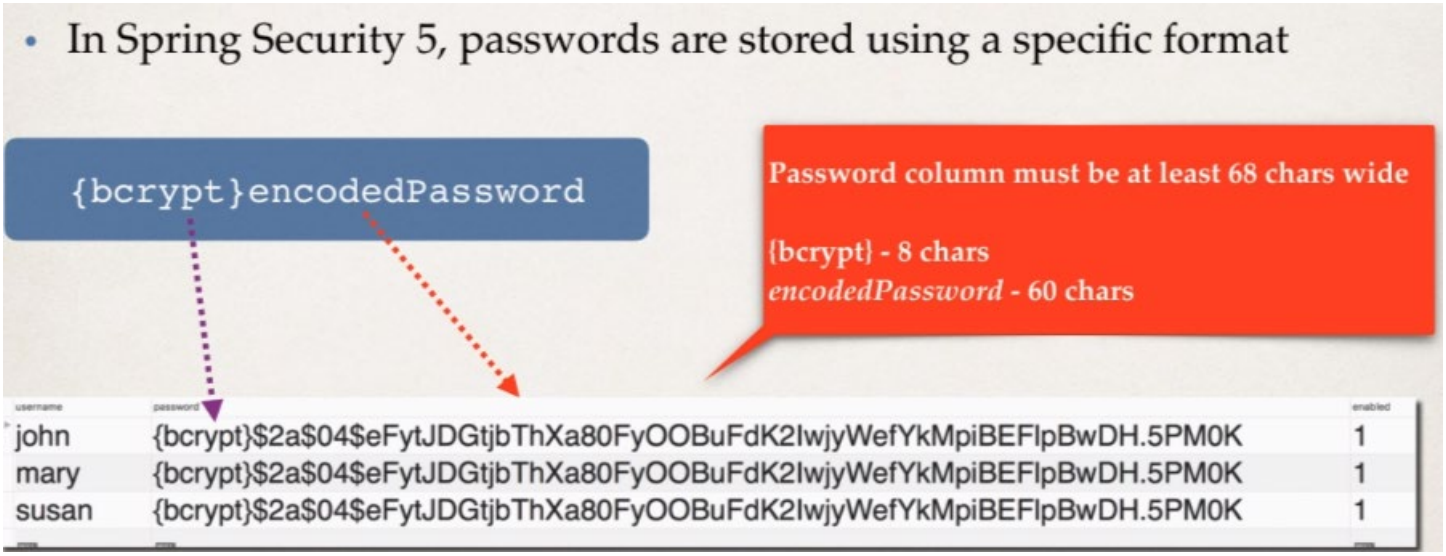


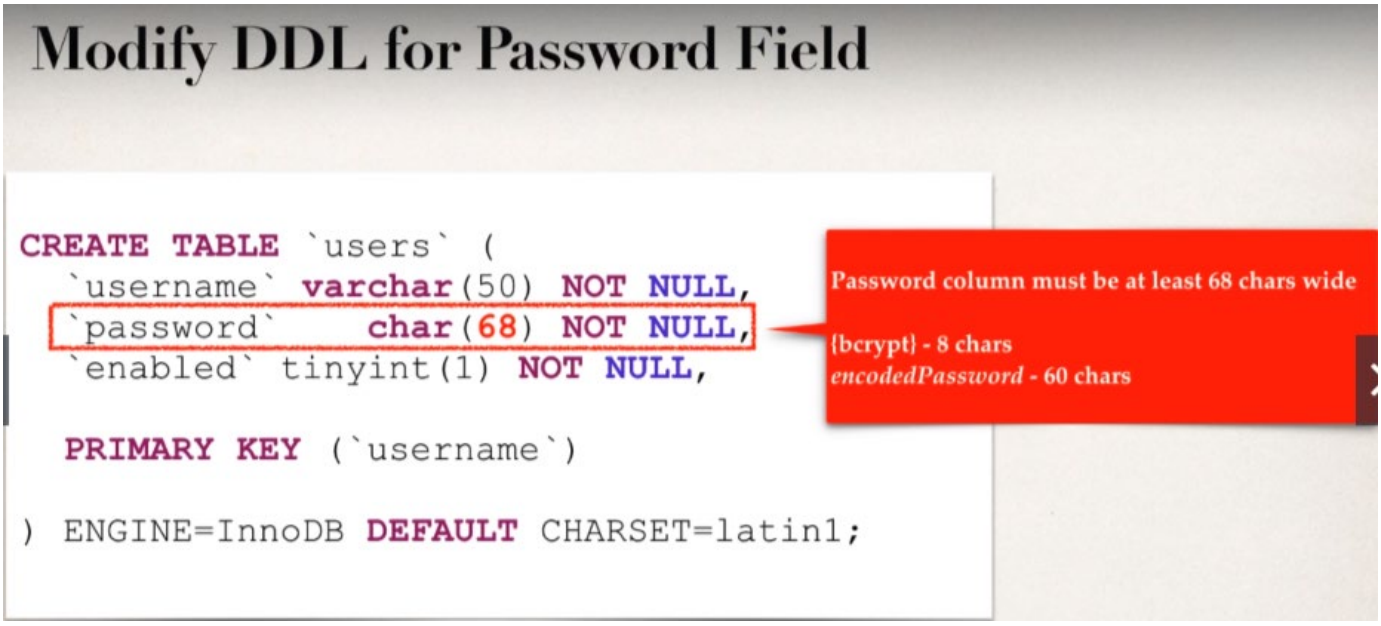
Now we can use these encrypted passwords and add them to our user accounts in our database.

- **Spring Security – Password Encryption – Spring Configuration**

Development Process:







Step 1: Develop SQL Script to setup database tables

The encoding algorithm id

The encrypted password: fun123

```
INSERT INTO `users`
VALUES
('john','{bcrypt}$2a$04$eFytJDGtjbThXa80FyOOBuFdK2IwjyWefYkMpiBEFlpBwDH.5PM0K',1),
```

Let's Spring Security know the passwords are stored as encrypted passwords: bcrypt

Step 2: Point to New Database Schema

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

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

#
# Connection pool properties
#
```

```
CREATE DATABASE  IF NOT EXISTS `spring_security_demo_bcrypt`;
USE `spring_security_demo_bcrypt`;
```

Spring Security Login Process

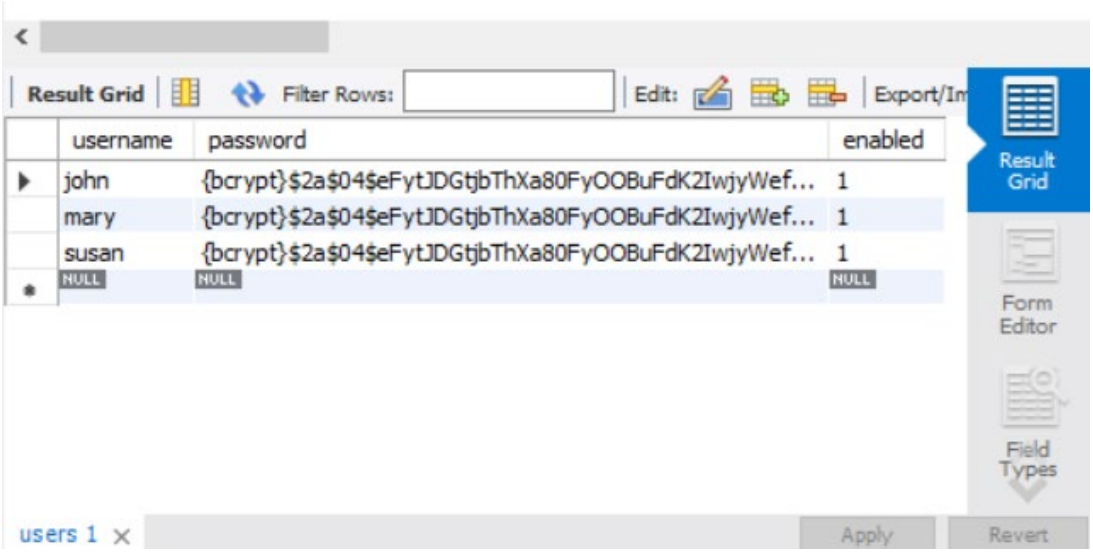
Note:
The password from db is NEVER decrypted

Because bcrypt is a one-way encryption algorithm

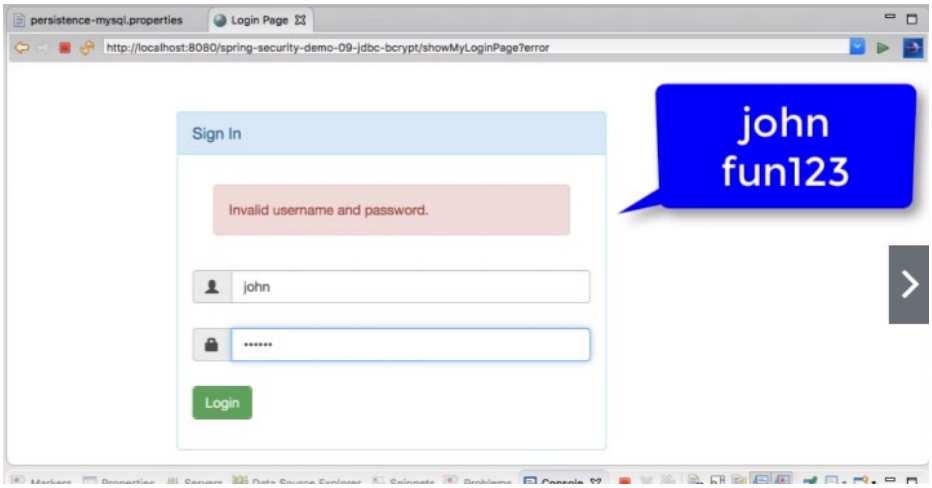
1. Retrieve password from db for the user
2. Read the encoding algorithm id (bcrypt etc)
3. For case of bcrypt, encrypt plaintext password from login form (using salt from db password)
4. Compare encrypted password from login form WITH encrypted password from db
5. If there is a match, login successful
6. If no match, login NOT successful

Download the **SQL Scripts** from the link

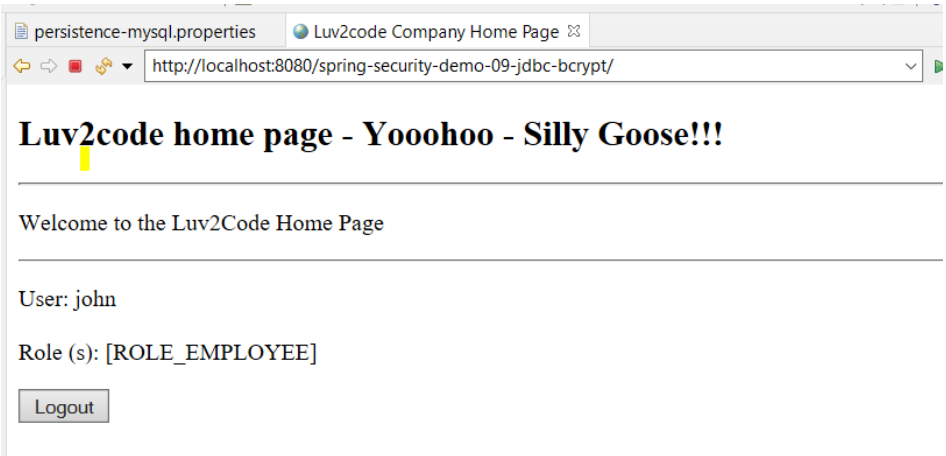
Resultant table:



Output:



So, this is working as "**john**" being an employee is able to login



Bonus Lecture: Spring Security - Adding a Public Landing Page

Question
Hello chad! I want my application to have a landing page that is accessible to everyone at first, the user can then login to the application. Our current framework only points to the login page. How do i implement this scenario?

Answer
Good question!

You can add a public view page and set up the security constraints to allow access to the view page. In this example, we have a view page that anyone can access. then they can click the link to access the secure pages.

Here is the solution code for the project: **spring-security-landing-page.zip**

This project has the following modifications.

1. Updated security configs to allow public access to landing page
2. Updated controller to send requests to landing page
3. New landing page

Details below --

1. Updated security configs to allow public access to landing page

See the config below. It will "permit all" access to the landing page "/".

```
@Override
    protected void configure(HttpSecurity http) throws Exception {

            http.authorizeRequests()
                    .antMatchers("/").permitAll(). // allow public access to landing page
                    .antMatchers("/employees").hasRole("EMPLOYEE")
                    .antMatchers("/leaders/**").hasRole("MANAGER")
                    .antMatchers("/systems/**").hasRole("ADMIN")
                    .and()
                    .formLogin()
```

```
                        .loginPage("/showMyLoginPage")
                        .loginProcessingUrl("/authenticateTheUser")
                        .permitAll()
                .and()
                .logout()
                        .logoutSuccessUrl("/")  // after logout redirect to landing page (root)
                        .permitAll();

    }
```

2. Updated controller to send requests to landing page
In the controller file, added new "/" mapping to send to landing page. And changed the original home mapping to "/employees". see changes in bold.

File: DemoController.java

```
@Controller
public class DemoController {

        @GetMapping("/")
        public String showLanding() {

                return "landing";
        }
...
}
```

3. New landing page
Created a new view page for landing information. Anyone can access this page

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

```
<html>

<head>
        <title>luv2code Landing Page</title>
</head>

<body>
        <h2>luv2code Landing Page</h2>
        <hr>

        <p>
        Welcome to the landing page!  This page is open to the public ... no login required :-)
        </p>

        <hr>

        <p>
                <a href="${pageContext.request.contextPath}/employees">Access Secure Site (requir
    es login)</a>
        </p>

</body>

</html>
```

Now when you run the application, we have a view page that anyone can access. then they can click the link to access the secure pages.