
UNIT 13 CUSTOM LOGIN USING SECURITY

Structure

- 13.0 Introduction
- 13.1 Objectives
- 13.2 Custom Login form creation
- 13.3 Spring Config for Custom Login Form
- 13.4 Create Request mapping and building Custom Login Form
- 13.5 Testing Custom Login Form
 - 13.5.1 Integration Test in Spring Boot
- 13.6 Adding Logout Support
- 13.7 Summary
- 13.8 Solutions/ Answer to Check Your Progress
- 13.9 References/Further Reading

13.0 INTRODUCTION

Unit 12 explained how to secure the spring application using spring security. The built-in login module of Spring Security authenticates the users and provides accessibility to the application. Default spring security does not require to create new jsp page for login since built-in login module already has default login page, which is used to authenticate the user. The default login page provided by spring security is not suitable for enterprise web application since the developer does not have control on designing and processing behavior. This unit describes how to customize the login page into spring security. The default login page, provided by Spring Security, has been used in the example written in unit 12, section 12.8. In this unit, the example executed in section 12.8 will be incorporated with a custom login page.

Testing is a very important phase of any application to ensure whether it works as per expected functionality or not. Integration testing and unit testing play a vital role in making the application error prone. This unit describes the annotation available in spring boot for Integration testing. The next unit will explain about unit test annotation available in spring boot.

Once user finishes all the activity on a web portal, he must be provided with a way to invalidate the session using logout functionality. Spring logout feature with various available methods has been explained in this unit.

13.1 OBJECTIVES

After going through this unit, you should be able to:

- ... create custom login page for spring security,
- ... configure the custom login page,
- ... execute the spring boot application with custom login page spring security, and
- ... develop logout support in Spring Applications.

13.2 CUSTOM LOGIN FORM CREATION

If no login form is configured into spring security, spring boot provides default login screen for spring security. Default login screen will be used to authenticate the user, and user will be allowed to access a resource. In an enterprise application default login screen is not suitable since enterprise does not have full control over design, processing URLs etc.

Customized login form creation and configuration is very easy in spring security. Custom login form in Spring Application is nothing but a simple html or jsp, like other web pages in java. A custom login form enables an enterprise to customize the login form as per their choice of design. It provides the developer to have full control over css and processing URL. A custom login form must be having below artifacts:

- ... Form action URL where the form is POSTed to trigger the authentication process
- ... Username input field for the username
- ... Password input field for the password

Sample custom login page is shown below. You can put the css styling as per application requirements. The following written custom login page will be used in the execution of the application into example.

Form Action URL(Custom login Controller):

```
package com.ignou.javabasedspringsecurity.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping(value="/customlogin")

public class LoginController
{
    @GetMapping

    public String login()
    {
        return "customlogin";
    }
}
```

Custom Login Form with CSS:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width, initial-scale=1">
<style>
body {font-family: Arial, Helvetica, sans-serif;}
form {border: 3px solid #f1f1f1; width: 50%; margin: auto;}

input[type=text], input[type=password]
```

```
{
  width: 100%;
  padding: 12px 20px;
  margin: 8px 0;
  display: inline-block;
  border: 1px solid #ccc;
  box-sizing: border-box;
}

button
{
  background-color: #4CAF50;
  color: white;
  padding: 14px 20px;
  margin: 8px 0;
  border: none;
  cursor: pointer;
  width: 100%;
}

button:hover
{
  opacity: 0.8;
}

.cancelbtn
{
  width: auto;
  padding: 10px 18px;
  background-color: #f44336;
}

.container
{
  padding: 16px;
  width: 50%;
  margin: auto;
}

.errormsg
{
  width: 100%;
  color: red;
  padding: 12px 0px;
  text-align: center;
}

span.psw
{
  float: right;
  padding-top: 16px;
}

</style>
</head>
<body>

<h2 style="text-align: center;">Spring Security Custom Login Form</h2>

<form action="/signin" method="post">
<div class="container">
  <c:if test="${param.error ne null}">
    <div class="errormsg"><b>Invalid
credentials</b></div>
  </c:if>
  <label for="uname"><b>Username</b></label>
  <input type="text" placeholder="Enter Username" name="username" required>

  <label for="psw"><b>Password</b></label>
  <input type="password" placeholder="Enter Password" name="password" required>

  <button type="submit">Login</button>
  <label>
  <input type="checkbox" checked="checked" name="remember"> Remember me
  </label>
</div>

<div class="container">
  <button type="button" class="cancelbtn">Cancel</button>
  <span class="psw">Forgot <a href="#">password?</a></span>
```

```

</div>
</form>

</body>
</html>

```

The login form created above has the following relevant artifacts:

- ... /signin – the URL where the form is POSTed to trigger the authentication process
- ... username – input field name for the username
- ... password – input field name for the password

Above jsp outputs the following login screen (Figure 13.1).

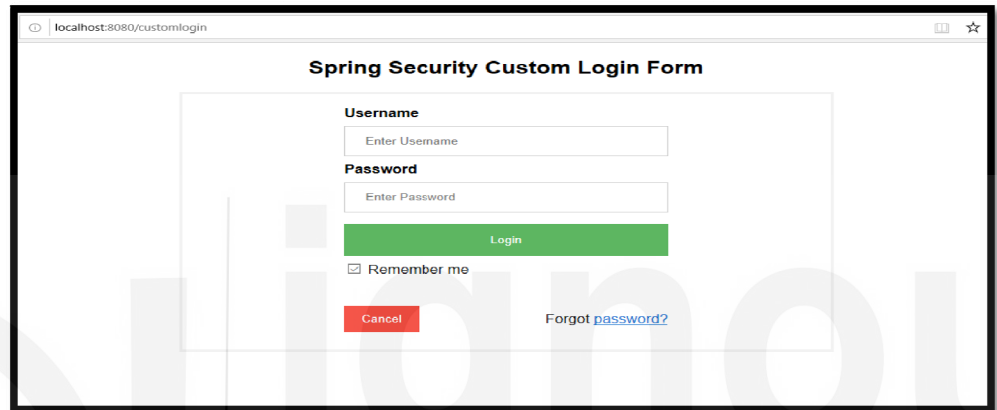


Figure13.1: Custom Login Page in Spring Security

13.3 SPRING Security CONFIG with Spring MVC FOR CUSTOM LOGIN FORM

Once you create the custom login page, you need to configure the custom login page in spring security. There is a detailed explanation regarding Java-based Spring Security configuration in section 12.5. Configuration for custom login page in spring security is very easy. You just need to set the loginPage in FormLoginConfigurer object provided by formLogin(). Sample Spring security configuration with custom login page is shown below.

```

package com.ignou.javabasedspringsecurity.security;
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter
{
    @Autowired
    PasswordEncoderpasswordEncoder;

    @Bean
    public PasswordEncoderpasswordEncoder ()
    {
        return new BCryptPasswordEncoder();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception
    {
        auth.inMemoryAuthentication().passwordEncoder(passwordEncoder)
            .withUser("testuser")
            .password(passwordEncoder.encode("user@123")).roles("USER").and()
            .withUser("testadmin").password(passwordEncoder.encode("admin@123"))
            .roles("USER", "ADMIN");
    }
}

```

```
protected void configure(HttpSecurity http) throws Exception
{
    http.csrf().disable()
    .authorizeRequests()
    .antMatchers("/customlogin").permitAll()
    .anyRequest().authenticated()
    .and()
    .formLogin()
    .loginPage("/customlogin")
    .loginProcessingUrl("/signin")
    .defaultSuccessUrl("/", true)
    .failureUrl("/customlogin?error=true");
}
```

In the above configuration, in-memory authentication has been configured with two users as testuser with Role USER and testadmin user with Role as USER, ADMIN both. The following section will explain the elements used to create the form login configuration.

authorizeRequests()

`<intercept-url>` element with `access="permitAll"` configures the authorization to **allow** all the requests on that particular path. Thus the requests for which no authorization is required, can be achieved without disabling the spring security using `permitAll()`.

formLogin()

There are several methods to configure the behavior of `formLogin()`.

- ... `loginPage()` – the custom login page
- ... `loginProcessingUrl()` – the url to submit the username and password to
- ... `defaultSuccessUrl()` – the landing page after a successful login
- ... `failureUrl()` – the landing page after an unsuccessful login

Check Your Progress 1

- 1) Why is login form customization required in an application? Explain the customized login form configuration with sample code.

.....

.....

.....

.....

- 2) Define `loginPage()` and `loginProcessingUrl()` in Spring Boot Security configuration.

.....

.....

.....

.....

13.4 CREATE REQUEST MAPPING AND BUILDING CUSTOM LOGIN FORM

Custom login form creation has been explained in Section 13.2. A very simple custom login form named as `customlogin.jsp` without any css is shown below

```

<html>
<body>
<div>
<center>Custom Spring Login Form</center>
<form action="/signin" method="post">
<input name="username" type="text" placeholder="Enter Username" />
<input name="password" type="text" placeholder="Enter password" />
<input type="submit" />
</form>
</div>
</body>
</html>

```

Custom login form must be mapped with controller in order to render the page to the user. The controller class is responsible for processing the request received from the dispatcher servlet. Spring provides annotation like `@Controller` to make a POJO class a controller in spring mvc application. `@RequestMapping` can be used at class level to map the controller which will be responsible for handling the request, and at the method level to map the method, which will be responsible for handling the requested URL. In previous code snippet for security configuration in section 13.3, `loginPage("/customlogin")` has been used. `"/customlogin"` should be mapped into a controller which will be responsible to process it and to return the corresponding view. Request mapping is shown below for `"/customlogin"` which return the view name as `customlogin`.

Spring view resolver resolves this view name as `customlogin.jsp` and it renders the jsp created into section 13.2

```

package com.ignou.javabasedspringsecurity.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping(value="/customlogin")
public class LoginController
{

    @GetMapping
    public String login()
    {
        return "customlogin";
    }

}

```

In above code snippet, `LoginController` is mapped with the URL pattern `/customlogin` and `login()` method is returning view name as `cutomlogin`. View resolver will resolve the view and corresponding view will be rendered.

13.5 TESTING CUSTOM LOGIN FORM

This section describes the execution of spring boot application with custom login form using spring security. Configuration code and request mapping code has been written

in code sample. This section also describes the annotations available in spring boot for unit testing.

Directory Structure for Spring Project in eclipse will be as below (Figure 13.2) –

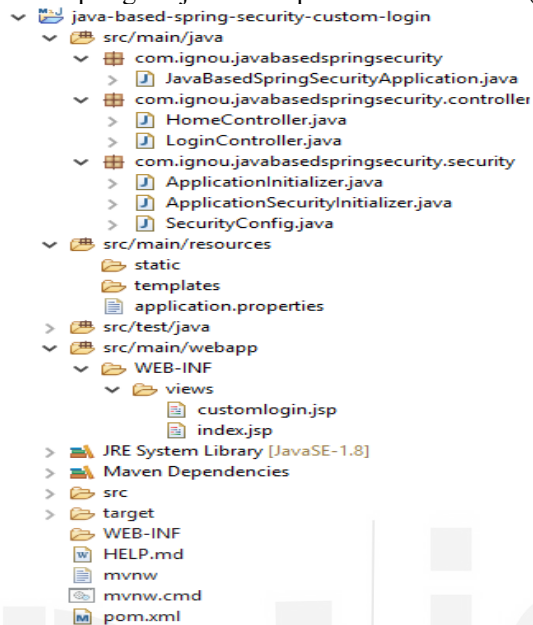


Figure 13.2: Project Structure In Eclipse

Spring Security Configuration for custom login form

```
package com.ignou.javabasedspringsecurity.security;

@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    PasswordEncoder passwordEncoder;
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication().passwordEncoder(passwordEncoder).withUser("testuser")
            .password(passwordEncoder.encode("user@123")).roles("USER").and().withUser("testadmin")
            .password(passwordEncoder.encode("admin@123")).roles("USER", "ADMIN");//(1)
    }
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
            .authorizeRequests()
            .antMatchers("/customlogin*").permitAll() //(2)
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/customlogin") //(3)
            .loginProcessingUrl("/signin") //(4)
            .defaultSuccessUrl("/", true)
            .failureUrl("/customlogin?error=true");
    }
}
```

In above code followings have been configured.

- 1) In-Memory authentication is configured for users testuser and testadmin.

- 2) /customlogin URL should be accessible to all. Thus permitAll() will allow all request to pass.
- 3) formLogin().loginPage("/customlogin") configures the customlogin page. Controller must have a mapping corresponding to Url pattern "/customlogin" to return customlogin page.
- 4) Custom login form processing URL is configured by loginProcessingUrl("/signin"). Custom login page form action must be "/signin"

Controller Classes and View

```
package com.ignou.javabasedspringsecurity.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping(value="/customlogin")
public class LoginController
{

    @GetMapping
    public String login()
    {
        return "customlogin";
    }
}

package com.ignou.javabasedspringsecurity.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
@RequestMapping(value="/")
public class HomeController
{
    @GetMapping
    public ModelAndView index()
    {
        ModelAndView mv = new ModelAndView();
        mv.setViewName("index");
        mv.getModel().put("message", "User, Spring Security with custom login page implemented Successfully !!");
        return mv;
    }
}
```

Custom login form with CSS

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width, initial-scale=1">
<style>
body {font-family: Arial, Helvetica, sans-serif;}
form {border: 3px solid #f1f1f1; width: 50%; margin: auto;}

input[type=text], input[type=password]
{
    width: 100%;
    padding: 12px 20px;
    margin: 8px 0;
```



```

display: inline-block;
border: 1px solid #ccc;
box-sizing: border-box;
}

button
{
    background-color: #4CAF50;
    color: white;
    padding: 14px 20px;
    margin: 8px 0;
    border: none;
    cursor: pointer;
    width: 100%;
}

button:hover
{
    opacity: 0.8;
}

.cancelbtn
{
    width: auto;
    padding: 10px 18px;
    background-color: #f44336;
}

.container
{
    padding: 16px;
    width: 50%;
    margin: auto;
}

.errormsg
{
    width: 100%;
    color: red;
    padding: 12px 0px;
    text-align: center;
}

span.psw
{
    float: right;
    padding-top: 16px;
}

</style>
</head>
<body>

<h2 style="text-align: center;">Spring Security Custom Login Form</h2>

<form action="/signin" method="post">
<div class="container">
    <c:if test="${param.error ne null}">
        <div class="errormsg"><b>Invalid
credentials</b></div>
    </c:if>
    <label for="uname"><b>Username</b></label>
    <input type="text" placeholder="Enter Username" name="username" required>

    <label for="psw"><b>Password</b></label>
    <input type="password" placeholder="Enter Password" name="password" required>

    <button type="submit">Login</button>
    <label>
    <input type="checkbox" checked="checked" name="remember"> Remember me
    </label>
</div>

<div class="container">
    <button type="button" class="cancelbtn">Cancel</button>
    <span class="psw">Forgot <a href="#">password?</a></span>
</div>
</form>

</body>

```

</html>

Execute the application as explained in Unit 12. In above example spring security has been configured with User as testuser with credential as user@123 and another User as testadmin with credential as admin@123.

Output:

If user tries to access localhost:8080 and user is not logged in, spring will redirect the user on localhost:8080/customlogin and custom login page will be rendered as shown below in Figure 13.2:

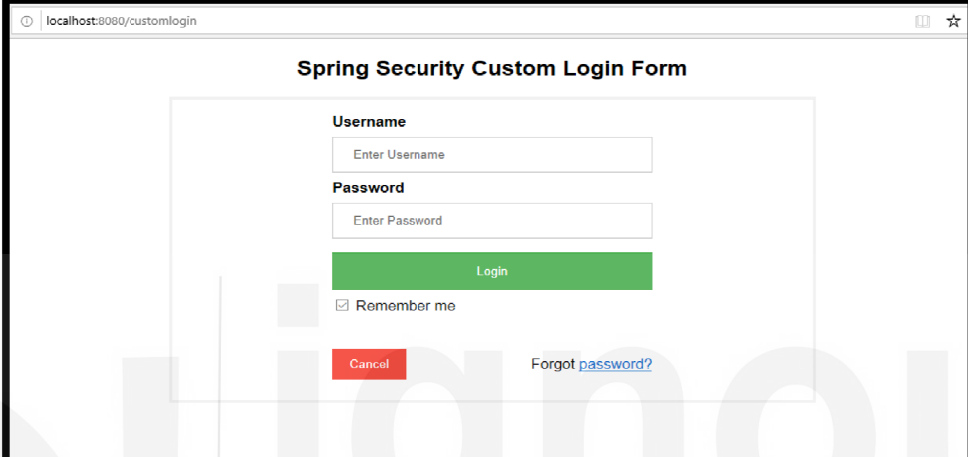
A screenshot of a web browser window showing the 'Spring Security Custom Login Form'. The browser's address bar displays 'localhost:8080/customlogin'. The form is centered on the page and contains the following elements: a title 'Spring Security Custom Login Form', a 'Username' label above a text input field with placeholder text 'Enter Username', a 'Password' label above a text input field with placeholder text 'Enter Password', a green 'Login' button, a checked 'Remember me' checkbox, a red 'Cancel' button, and a link 'Forgot password?'. The form is enclosed in a light gray border.

Figure 13.3: Custom Login Page In Spring Security

If user provides wrong credentials, alert from spring security will be as shown in Figure 13.4 :

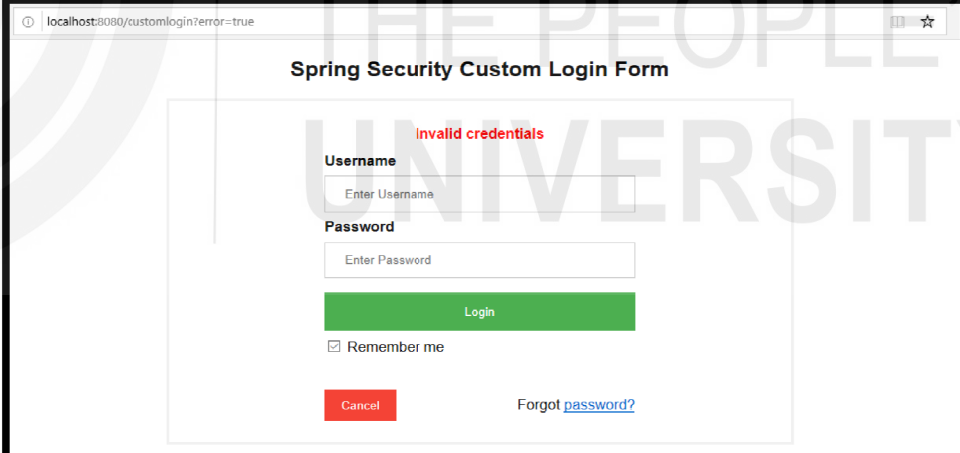
A screenshot of a web browser window showing the 'Spring Security Custom Login Form' after an invalid login attempt. The browser's address bar displays 'localhost:8080/customlogin?error=true'. The form is identical to Figure 13.3, but with an additional red error message 'Invalid credentials' displayed above the 'Username' input field. The 'Login' button remains green, and the 'Remember me' checkbox is still checked.

Figure 13.4: Invalid Credentials Error Message

After Successful login user will be redirected to index.jsp and it will output as shown in Figure 13.5.

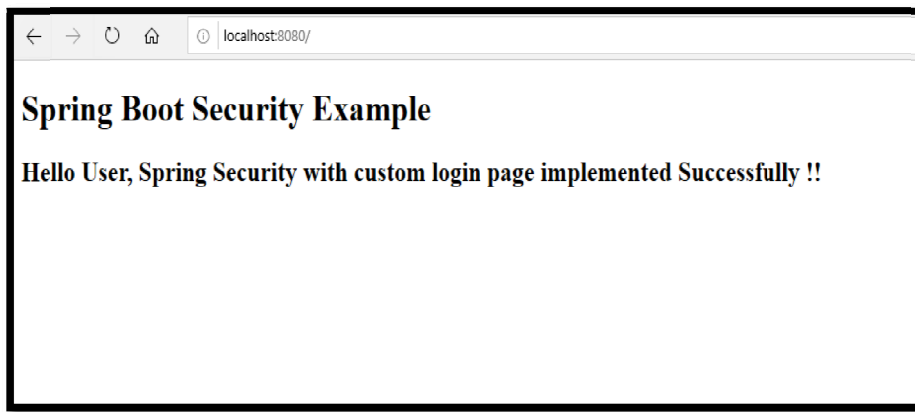


Figure 13.5: Success Screen After Authentication

13.5.1 Integration Test in Spring Boot

The following section describes the integration test in the spring boot application. Before getting dive into integration testing, let us define how integration test is different from unit test.

Unit testing aims at individual modules of an application without any interaction with dependencies to validate whether code is working as per the expected result or not. Integration testing aims to validate the code is working fine when different modules are combined as a group.

Previous sections described custom login form integration with spring security. Here integration test will be described for configured custom login form with spring security. Spring provides many annotations to perform integration test.

Spring Security and MockMvc Set Up

Spring Security with Spring MVC test can be used with the configuration of Spring Security FilterChainProxy as a Filter. The following set up performs the same.

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
class JavaBasedSpringSecurityApplicationTests
{
    @Autowired
    private MockMvc mockMvc;
}
```

@RunWith: `@RunWith(SpringRunner.class)` instructs JUnit to run using Spring's testing support. `SpringRunner` is the new name for `SpringJUnit4ClassRunner`.

@SpringBootTest: This annotation instructs Spring Boot to use main class annotated with `@SpringBootApplication` to start a Spring application context. Spring boot application testing requires nothing special to do. It's **ApplicationContext** that needs to be available to test Spring Boot application. `@RunWith` instructs the spring-test module to create an `ApplicationContext`. `SpringBootTest` loads complete application and injects all the beans, which can be slow.

@AutoConfigureMockMvc: Spring provides a way to simulate the HTTP request in which application code will be executed exactly the same way as it were processing a

real HTTP request but without any server start. For this simulation use, Spring's MockMvc and that can be configured using `@AutoConfigureMockMvc`.

SecurityMockMvcRequestPostProcessors

RequestPostProcessor interface is provided by Spring MVC Test. These request processors can be used to modify the request. Spring Security has provided many implementations of RequestPostProcessor to make testing easy and efficient. Spring Security's RequestPostProcessor implementations can be used by static import of as below:

```
import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.*;
```

There are two ways to run the test cases as a specific User in Spring MVC Test.

- ... Running the test case as a specific user with RequestPostProcessor
- ... Running the test case as a specific user with Annotations

Running the test case as a user with RequestPostProcessor

The following test will run as a specific user, which need not to exist since we are mocking the user, with the username "user", the password "password", and the role "ROLE_USER":

```
@Test
public void testAlreadyLoggedInUser() throws Exception
{
    mvc.perform(get("/").with(user("user"))).andExpect(authenticated().withUsername("user"))
        .andExpect(authenticated().withRoles("USER"))
        .andExpect(model().attribute("message",
            "User, Spring Security with custom login page implemented Successfully !!"));
}
```

In the set up MockMvc is autoconfigured as mvc and it is being used to simulate the HTTP request. Here RequestPostProcessoruser() has been used to modified the get("/") request. User RequestPostProcessor can be customized for password and roles as –

```
mvc.perform(get("/").with(user("John").password("pass@123").roles("USER","ADMIN")))
```

Test can be run as an anonymous user with following configuration.

```
mvc.perform(get("/").with(anonymous()))
```

Running the test case as a user with Annotation

Instead of RequestPostProcessor to create a user, annotation can be used for the same. The following test will run as a specific user, which does not need to exist since we are mocking the user, with the username "user", the password "password", and the role "ROLE_USER":

```
@Test
@WithMockUser
public void testAlreadyLoggedInUser() throws Exception
{
    mvc.perform(get("/").andExpect(authenticated().withUsername("user"))
        .andExpect(authenticated().withRoles("USER"))
        .andExpect(model().attribute("message",
            "User, Spring Security with custom login page implemented Successfully !!"));
}
```

@WithMockUser can be customized with the following attributes:

```
... username: String
... authorities: String[]
... password: String
... roles: String[]
```

SecurityMockMvcRequestBuilders

RequestBuilder interface is provided by Spring MVC Test. These request builders can be used to create the MockHttpServletRequest into the test. Few implementations of RequestBuilder has been provided by Spring Security to make testing easy and efficient. Spring Security's RequestBuilder implementations can be used by static import of as below:

```
import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestBuilders.*;
```

Form Based Authentication Testing

Form based authentication can be tested very easily in Spring Boot with the help of SecurityMockMvcRequestBuilders. Below will submit a POST request to “/login” with the username as “user”, the password as “password” and valid CSRF token.

```
mvc.perform(formLogin())
```

Customization of formLogin() is very easy and an example of it is given below.

```
@Test
public void testFormLogin() throws Exception
{
    mvc.perform(formLogin("/signin").user("testadmin").password("admin@123").andExpect(
        status().isFound())
        .andExpect(redirectedUrl("/")).andExpect(authenticated().withUsername("testadmin"))
        .andExpect(authenticated().withRoles("ADMIN", "USER")));
}
```

In above configuration a POST request for “/signin” with the username as “testadmin” and the password as “admin@123” with a valid CSRF token will be submitted. Parameters name can be customized as follows -

```
mvc.perform(formLogin("/signin").user("uname","admin@gmail.com").password("password","admin@"))
```

Logout Testing Using SecurityMockMvcRequestBuilders

Logout functionality can be tested very easily using SecurityMockMvcRequestBuilders. Example for Logout test is as followings-

```
mvc.perform(logout())
```

Logout processing URL can be customized as

```
mvc.perform(logout("/signout"))
```

Check the folder src/test for unit test cases. All unit test cases can be executed by right click on the project -> Run As -> JUnit Test as shown below screenshot in Figure 13.6

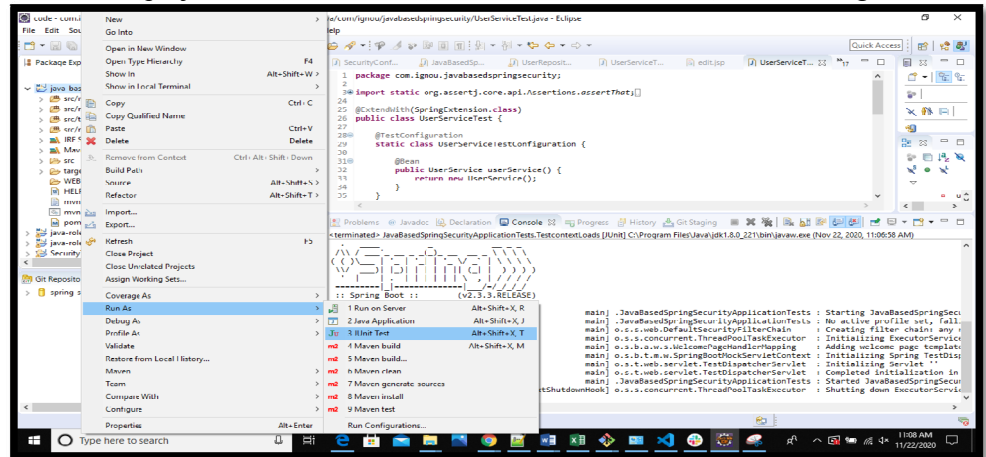


Figure 13.6: UnitCase Execution From Eclipse IDE

Successful execution of Unit test case will output the following. As you can see that all test cases are passed that is shown by green tick and green progress bar in the Figure 13.7.

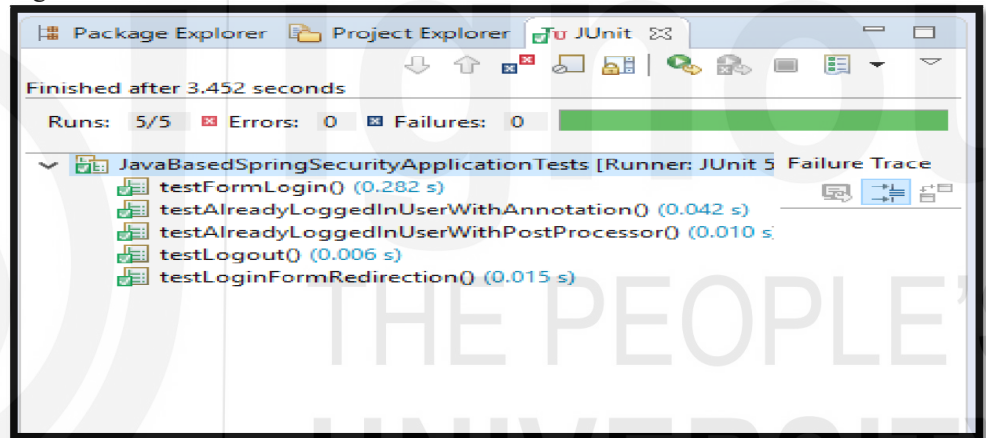


Figure 13.7: Unit Test Cases Execution Log

13.6 ADDING LOGOUT SUPPORT

The basic configuration of Spring Logout functionality using the logout() method is simple enough:

```
@Configuration
@EnableWebSecurity
public class SecSecurityConfig extends WebSecurityConfigurerAdapter
{
    @Override
    protected void configure(final HttpSecurity http) throws Exception
    {
        http
            //...
        .logout()
            //...
    }
    //...
}
```

The above configuration enables default logout mechanism with processing url: /logout, which used to be /j_spring_security_logout before Spring Security 4. We can

customize the logout feature in spring security. Let us see the advanced customization of spring logout.

logoutSuccessUrl()

On successfully logout operation, Spring Security will redirect the user to a specified page. By default, this is the root page ("/"), but this is configurable:

```
//...
.logout()
.logoutSuccessUrl("/successlogout")
//...
```

logoutUrl()

Like other defaults in Spring Security, logout mechanism has a default processing URL as well – /logout. To hide the information regarding which framework has been used to secure the application, it's better to change the default value. It can be done as below:

```
// ...
.logout()
.logoutUrl("/do_logout")
// ...
```

invalidateHttpSession and deleteCookies

These two advanced attributes control the session invalidation as well as a list of cookies to be deleted when the user logs out. Default value for invalidateHttpSession is true on logout.

```
//...
.logout()
.logoutUrl("/do_logout")
.invalidateHttpSession(true)
.deleteCookies("JSESSIONID")
// ...
```

Logout feature can be tested by enabling the logout option for the user. Add the logout option by doing the changes into index.jsp as shown below.

```
<!DOCTYPEhtml>
<%@taglibprefix="spring"uri="http://www.springframework.org/tags"%>
<html1lang="en">
<body>
    <div>
        <div>
            <astyle="text-align: left;" href="/Logout">logout</a>
            <h1>Spring Boot Security Example</h1>
            <h2>Hello ${message}</h2>
        </div>
    </div>
</body>
</html>
```

Logout configuration into WebSecurityConfig.java is described below.

```

package com.ignou.javabasedspringsecurity.security;

@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter
{
    @Autowired
    PasswordEncoder passwordEncoder;
    @Bean
    public PasswordEncoder passwordEncoder()
    {
        return new BCryptPasswordEncoder();
    }
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception
    {
        auth.inMemoryAuthentication().passwordEncoder(passwordEncoder).withUser("testuser")
            .password(passwordEncoder.encode("user@123")).roles("USER").and().withUser("testadmin")
            .password(passwordEncoder.encode("admin@123")).roles("USER", "ADMIN");
    }
    protected void configure(HttpSecurity http) throws Exception
    {
        http.csrf().disable()
            .authorizeRequests()
            .antMatchers("/customlogin*").permitAll()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/customlogin")
            .loginProcessingUrl("/signin")
            .defaultSuccessUrl("/", true)
            .failureUrl("/customlogin?error=true")
            .and()
            .logout();
    }
}

```

Execute the application and after successful login you will see logout option as shown below in Figure 13.8:

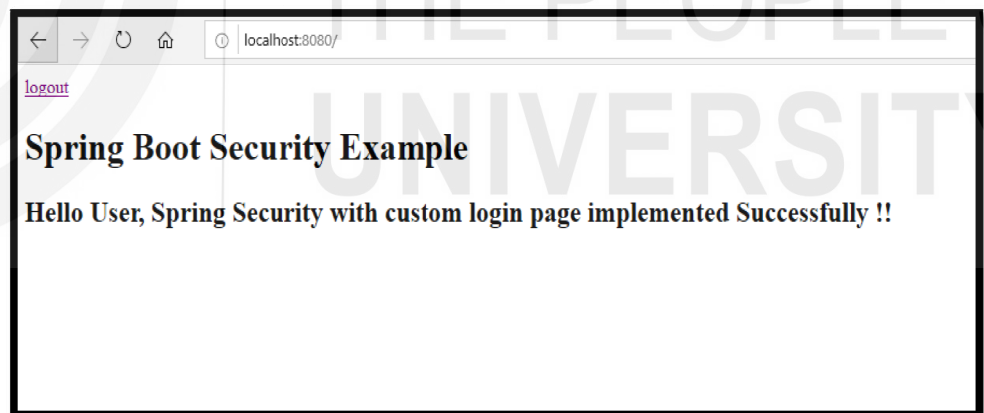


Figure 13.8: Success Screen with Logout Feature

After clicking logout link, user will be logged out and redirected to login page as shown in Figure 13.9

Spring Security Custom Login Form

Username

Enter Username

Password

Enter Password

Login

☒ Remember me

Cancel

Forgot [password?](#)

Figure 13.9: Success Screen with Logout Feature

Check Your Progress 2

- 1) Explain the usage of `@SpringBootTest`.
.....
.....
.....
.....
- 2) Write the unit test case to execute it as a user with `RequestPostProcessor` for the URL pattern `"/"` which returns model attribute with key as `"message"` and value as `"Hello World!!"`
.....
.....
.....
.....
- 3) Write the unit test case to execute it as a user with Annotation for the URL pattern `"/"` which returns model attribute with key as `"message"` and value as `"Hello World!!"`
.....
.....
.....
.....
- 4) Write Unit test case to test the custom login form having `"/signin"` as `loginProcessingUrl` and `"/"` as `defaultSuccessUrl`.
.....
.....
.....
.....

13.7 SUMMARY

Default Spring Security login form is not suitable for enterprises application since developers do not have full control over css and form design. Spring Security provides a way to configure custom login form in Spring Application. This unit has explained the configuration of custom login form with an example and its execution.

Unit test plays a vital role to ensure the correctness of code, and it identifies every defect that may arise before code is integrated with other modules. Many Spring Boot annotations available for Integration testing, such as `@SpringBootTest`, `@AutoConfigureMockMvc`, `@WithMockUser` has been explained.

Logout feature is an important feature from security perspective. At the end of this unit, Spring provided logout support has been explained with configurable properties.

13.8 SOLUTIONS/ANSWERS TO CHECK YOUR PROGRESS

☛ Check Your Progress 1

- 1) By default, spring boot provides default login screen for spring security. In enterprise application using default login screen, enterprise may not have full control over design, processing URLs etc. Custom login form in Spring Application is nothing but a simple html or jsp, like others web pages in java. Sample custom login form is shown below-

```
<html>
<body>
<div>
<center>Custom Spring Login Form</center>
<form action="/signin" method="post">
<input name="username" type="text" placeholder="Enter Username" />
<input name="password" type="text" placeholder="Enter password" />
<input type="submit" />
</form>
</div>
</body>
</html>
```

Configuration for custom login page in spring security is very easy. We just need to set the `loginPage` in `FormLoginConfigurer` object provided by `formLogin()`. Sample Spring security configuration with custom login page is shown below.

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter
{
    @Autowired
    PasswordEncoder passwordEncoder;
    @Bean
```

```

public PasswordEncoder passwordEncoder()
{
    return new BCryptPasswordEncoder();
}
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
    auth.inMemoryAuthentication().passwordEncoder(passwordEncoder).withUser("testuser")
        .password(passwordEncoder.encode("user@123")).roles("USER").and().withUser("testadmin")
        .password(passwordEncoder.encode("admin@123")).roles("USER", "ADMIN");
}
protected void configure(HttpSecurity http) throws Exception
{
    http.csrf().disable()
        .authorizeRequests()
        .antMatchers("/customlogin").permitAll()
        .anyRequest().authenticated()
        .and()
        .formLogin()
        .loginPage("/customlogin")
        .loginProcessingUrl("/signin")
        .defaultSuccessUrl("/", true)
        .failureUrl("/customlogin?error=true");
}
}

```

- 2) loginPage("/dologin") instructs Spring Security to perform following-
 - ... Whenever authentication is required, its being redirected to /dologin
 - ... Login page is being rendered when /dologin is requested
 - ... It will redired to /dologin?error(if default configuration is used for error) in the case of authentication failure.
 - ... On successful logout, it will redirect to /dologin?logout (In the case fo default configuration used)

loginProcessingUrl(/signin) instruct the Spring Security to validate the submitted credentials sent for /signinUrl. By default it redirects the user back to the page from where user came. Request will neither be passed to Spring MVC nor controller.

Check Your Progress 2

- 1) Spring-Boot provides @SpringBootTest annotation. This annotation provided spring boot feature in the test module and works by creating the ApplicationContext used in tests through SpringApplication. It starts the embedded server, creates a web environment and enables @Test methods to do integration testing.
By default, no server is started by @SpringBootTest. It provides several options to add webEnvironment regarding how test cases should be executed.
 - ... MOCK(Default): Loads a web ApplicationContext and provides a mock web environment
 - ... RANDOM_PORT: Loads a WebServerApplicationContext and provides a real web environment. The embedded server is started and listen to on a random port. This is the one that should be used for the integration test
 - ... DEFINED_PORT: Loads a WebServerApplicationContext and provides a real web environment.
 - ... NONE: Loads an ApplicationContext by using SpringApplication but does not provide any web environment

- 2) RequestPostProcessor interface is provided by Spring MVC Test. These request processors can be used to modify the request. Many implementations of RequestPostProcessor has been provided by Spring Security to make testing easy and efficient. Spring Security's RequestPostProcessor implementations can be used by static import of as below:

Import static

```
org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.*;
```

Sample code to execute a unit test case with RequestPostProcessor is as followings-

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
@AutoConfigureMockMvc
class JavaBasedSpringSecurityApplicationTests
{
    @Autowired
    private MockMvc mvc;
    @Test
    public void testAlreadyLoggedInUserWithPostProcessor() throws Exception
    {
        mvc.perform(get("/").with(user("user"))).andExpect(authenticated().withUsername("user"))
            .andExpect(authenticated().withRoles("USER"))
            .andExpect(model().attribute("message", "Hello World!!"));
    }
}
```

- 3) Sample code with Annotation is as below:

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
@AutoConfigureMockMvc
class JavaBasedSpringSecurityApplicationTests
{
    @Autowired
    private MockMvc mvc;
    @Test
    @WithMockUser()
    public void testAlreadyLoggedInUserWithAnnotation() throws Exception
    {
        mvc.perform(get("/").andExpect(authenticated().withUsername("user"))
            .andExpect(authenticated().withRoles("USER"))
            .andExpect(model().attribute("message", "Hello World!!")));
    }
}
```

- 4) Sample code with Annotation is as below:

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
@AutoConfigureMockMvc
class JavaBasedSpringSecurityApplicationTests
{
    @Autowired
    private MockMvc mvc;
    @Test
    public void testFormLogin() throws Exception
    {
        mvc.perform(formLogin("/signin").user("testadmin").password("admin@123"))
            .andExpect(status().isFound())
            .andExpect(redirectedUrl("/")).andExpect(authenticated().withUsername("testadmin"))
            .andExpect(authenticated().withRoles("ADMIN", "USER"));
    }
}
```

13.9 REFERENCES/FURTHER READING

- ... Craig Walls, “Spring Boot in action” Manning Publications, 2016.
(<https://doc.lagout.org/programmation/Spring%20Boot%20in%20Action.pdf>)
- ... Christian Bauer, Gavin King, and Gary Gregory, “Java Persistence with
Hibernate”,Manning Publications, 2015.
- ... Ethan Marcotte, “Responsive Web Design”, Jeffrey Zeldman Publication,
2011([http://nadin.miem.edu.ru/images_2015/responsive-web-design-2nd-
edition.pdf](http://nadin.miem.edu.ru/images_2015/responsive-web-design-2nd-edition.pdf))
- ... Tomcy John, “Hands-On Spring Security 5 for Reactive Applications”,Packt
Publishing,2018
- ... [https://docs.spring.io/spring-
security/site/docs/4.2.19.RELEASE/guides/html5/form-javaconfig.html](https://docs.spring.io/spring-security/site/docs/4.2.19.RELEASE/guides/html5/form-javaconfig.html)
- ... <https://www.baeldung.com/spring-security-login>
- ... <https://www.baeldung.com/spring-boot-testing>
- ... [https://docs.spring.io/spring-
framework/docs/current/reference/html/testing.html](https://docs.spring.io/spring-framework/docs/current/reference/html/testing.html)



ignou
THE PEOPLE'S
UNIVERSITY