UNIT 14 ROLE BASED LOGIN

Structure

14.0 INTRODUCTION

Role-Based access control (RBAC) allows us to define the accessibility based on user's roles and permissions. The roles in RBAC refer to the levels of access that user has to the resource. Roles and permissions are used to make a resource secured from unauthorized access. RBAC allows to define what end-users can perform at both broad and granular levels. While using RBAC, resource access of the users is analyzed, and users are grouped into roles based on common responsibilities and needs. Each user into the system is assigned with one or more roles and one or more permissions to each role. For an example, an admin can have permission to create a new post and edit the already created post while an editor may have permission to edit the post which already exists.

This Unit explains how to restrict resource access based on roles using the Spring security. Most of the applications display the logged in user details. This unit describes the various approaches to get logged in user details in controllers. It also describes how to retrieve security related information at view layer in jsp.

The previous unit explained how to write integration test cases using annotations provided by Spring. Good unit test cases make the codebase auto deployable and production ready since codebase can be validated with the help of unit test cases. This unit describes the unit test cases with available annotations in Spring.

Cross site request forgery, also known as CSRF or XSRF, is a type of attack in which attackers trick a victim to make a request that utilizes their authentication or authorization. The victim's level of permission decides the impact of CSRF attack. In the end of this Unit, Cross Site Request Forgery (CSRF) has been explained with an example and the solution to avoid CSRF attack.

14.1 OBJECTIVES

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

- explain Spring Security core component,
- describe logged in user detail in controllers,
- use Spring Security Expressions,
- describe and use Authentication and Authorization information at view layer,
- apply Role-Based Access Control and restrict the access based on roles, and
- describe Cross Site Request Forgery (CSRF).

14.2 DISPLAY USER ID AND ROLES – OVERVIEW

Various core classes and interface available in spring security to get security context are outlined. Spring Security core components are -

- **SecurityContextHolder:** A Class which provide access to SecurityContext
- **SecurityContext:** An Interface having Authentication and defining the minimum-security information associated with request
- **Authentication:** Represents the **Principal** in Spring security-specific way. The Spring Security **Principal** can only be retrieved as an Object from Authentication and needs to be cast to the correct UserDetails instance as –

```
UserDetailsuserDetails = (UserDetails)
authentication.getPrincipal();
System.out.println("User has authorities: "+
userDetails.getAuthorities());
```

- **GrantedAuthority:** Having the application wide permissions granted to a principal
- **UserDetails:**UserDetailshasthe required information to build an Authentication object from application's DAOs or other sources of security data.

The following sections explain how to retrieve user details in Spring Security. There are various ways to access logged-in user information in the Spring. Few ways are described below.

14.2.1 User Details in a Controller

In a @Controller annotated bean, principal can be defined as a method argument, and the Spring framework will resolve it correctly.

```
@Controller
public class UserSecurityController
{

    @RequestMapping(value = "/username", method = RequestMethod.GET)
    @ResponseBody
    public String loggedInUserName(Principal principal)
    {
        return principal.getName();
    }
}
```

Instead of Principal, Authentication can be used. Authentication allows us to get the granted authorities using getAuthorities() method while Spring Security principal can only be retrieved as an Object and needs to be cast to the correct UserDetails instance. Sample code is as following-

```
@Controller
public class UserSecurityController
{
    @RequestMapping(value = "/username", method = RequestMethod.GET)
    @ResponseBody
    public String loggeInUserName(Authentication authentication)
    {
        return authentication.getName();
    }
}
```

User Details can be retrieved from HTTP request as -

```
@Controller
public class UserSecurityController
{

    @RequestMapping(value = "/username", method = RequestMethod.GET)
    @ResponseBody
    public String loggeInUserName(HttpServletRequest request)
    {
        Principal principal = request.getUserPrincipal();
        return principal.getName();
    }
}
```

User Details can be retrieved in a Bean as –

```
Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
String currentPrincipalName = authentication.getName();
```

14.2.2 Spring view layer security and User Details using JspTaglibs

Spring Security provides its own taglib for basic security support, such as retrieving security information and applying security constraints at view layer jsp. To use spring security features in jsp, you need to add the following tag library declaration into jsp-

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

The tags provided by spring security to access security information and to secure the view layer of the application are as below:

- Authorize Tag
- Authentication Tag
- Accesscontrollist Tag
- Csrfinput Tag
- CsrfMetaTags Tag

Authorize Tag: This tag supports to secure information based on user's role or permission to access a URL. Authorize tag support two types of attributes.

- access
- url

In an application, view layer might need to display certain information based on user's role or based on authentication state. An example for the same is shown below.

```
<security:authorize access="!isAuthenticated()">
  Login
</security:authorize>
<security:authorize access="isAuthenticated()">
```

```
  Logout
</security:authorize>
```

Further, we can also display certain information based on user role as-

```
<security:authorize access="hasRole('ADMIN')">
    Delete Users
</security:authorize>
```

Value for access can be any of Spring Security Expression as below-

- hasAnyRole('ADMIN','USER') returns true if the current user has any of the listed roles
- isAnonymous() returns true if the current principal is an anonymous user
- isRememberMe() returns true if the current principal is a remember-me user
- isFullyAuthenticated() returns true if the user is authenticated and is neither anonymous nor a remember-me user

Using url attribute in authorize tag, at view layer we can check weather user is authorized to send request to certain URL:

```
<security:authorizeurl="/inventoryManagement">
<a href="/inventoryManagement">Manage Inventory</a>
</security:authorize>
```

Authentication Tag: This tag is not used to implement security but it allows access to the current Authentication object stored in the security context. It renders a property of the object directly in the JSP. If the principal property of the Authentication is an instance of Spring Security's UserDetails object, then username and roles can be accessed as –

- <security:authentication property=”principal.username” />
- <security:authentication property=”principal.authorities” />

Accesscontrollist Tag: This tag is used with Spring Security's ACL module. It checks list of required permissions for the specified domains. It executes only if current user has all the permissions.

Csrfinput Tag: For this tag to work, csrf protection should be enabled in spring application. If csrf is enabled, spring security inserts a csrf hidden form input inside <form:form> tag. But in case if we want to use html <form></form>, we need to put csrfinput tag inside <form> to create CSRF token as below –

```
<form method=”post” action=”/some/action”>
<security:csrfInput />
                Name:<input type=”text” name=”username” />
…
</form>
```

CsrfMetaTags Tag: If we want to access CSRF token in javascript, we have to insert token as a meta tag. For this tag to work, csrf protection should be enabled in Spring Application.

```
<html>
<head>
<title>JavaScript with CSRF Protection</title>
<security:csrfMetaTags />
<script type=”text/javascript” language=”javascript”>
            var csrfParameter =
$(“meta[name=’_csrf_parameter’]”).attr(“content”);
            var csrfHeader = $(“meta[name=’_csrf_header’]”).attr(“content”);
            var csrfToken = $(“meta[name=’_csrf’]”).attr(“content”);
</script>
</head>
<body>
…
```

```
</body>
</html>
```

☞Check Your Progress 1

1) What are Authorities in Authentication object?

……

……

……

……

2) Write the code to get logged in user's username from HttpServletRequest in a controller.

……

……

……

……

3) Explain with an example to display username and roles on jsp.

……

……

……

……

14.3 ROLES BASED LOGIN EXAMPLE

This section describes role-based login using Spring Security. Role based login means redirecting users to different URLs upon successful login according to the assigned role to the logged-in user. The following section explains an example with three types of users as Admin, Editor and normal User. Once user successfully logged into the system, based on role user will be redirected to its own url as –

- **Admin** will be redirected to **/admin**
- **Editor** will be redirected to **/editor**
- **User** will be redirected to **/home**

In previous spring security configurations, the success URL on successful authentication was configured using ***defaultSuccessUrl(Sring s)***. With the use of Spring ***defaultSuccessUrl("/homePage"),*** every user will be redirected to homepage url irrespective of role. To have more control over the authentication success mechanism, **Spring Security** provides a component that has the direct responsibility of deciding what to do after a successful authentication – the **AuthenticationSuccessHandler**.

AuthenticationSuccessHandleris an Interface and Spring provides SimpleUrlAuthenticationSuccessHandler, which implement this interface. For role based login, you need to configure **successHandler(AuthenticationSuccessHandlersuccessHandler)** instead of ***defaultSuccessUrl(Sring s).*** For custom success handler, you have two approaches.

1. **Implement *AuthenticationSuccessHandler*** interface to provide custom success handler.

2. **Extend*SimpleUrlAuthenticationSuccessHandler*** class and override the handle() method to provide the logic.

The below code sample is for custom success handler using approach 2.

```
publicclassCustomSuccessHandlerextendsSimpleUrlAuthenticationSuccessHandler
{
        privateRedirectStrategyredirectStrategy = newDefaultRedirectStrategy();

@Override
        protectedvoidhandle(HttpServletRequestrequest, HttpServletResponseresponse,
        Authentication authentication)throwsIOException
        {
                String targetUrl = determineTargetUrl(authentication);

                if (response.isCommitted())
                {
                        System.out.println("Can't redirect");
                        return;
                }

                redirectStrategy.sendRedirect(request, response, targetUrl);
        }

        /*
         * This method extracts the roles of currently logged-in user and returns
         * appropriate URL according to his/her role.
         */
        protected String determineTargetUrl(Authentication authentication)
        {
                String url = "";
                Map<String, String>roleTargetUrlMap = new HashMap<>();
                roleTargetUrlMap.put("ROLE_USER", "/home");
                roleTargetUrlMap.put("ROLE_ADMIN", "/admin");
                roleTargetUrlMap.put("ROLE_Editor", "/editor");

                Collection<? extendsGrantedAuthority>authorities =
                authentication.getAuthorities();
                List<String>roles = newArrayList<String>();
                for (GrantedAuthoritya :authorities)
                {
                        roles.add(a.getAuthority());
                }

                if (isAdmin(roles))
                {
                        url = "/admin";
                }
                elseif (isEditor(roles))
                {
                        url = "/editor";
                }
                elseif (isUser(roles))
                {
                        url = "/home";
                }
                else
                {
                        url = "/accessDenied";
                }

                returnurl;
        }

        privatebooleanisUser(List<String>roles)
        {
                if (roles.contains("ROLE_USER"))
                {
                        returntrue;
                }
                returnfalse;
        }

        privatebooleanisAdmin(List<String>roles)
        {
                if (roles.contains("ROLE_ADMIN"))
```

```
                {
                        returntrue;
                }
                returnfalse;
        }

        privatebooleanisEditor(List<String>roles)
        {
                if (roles.contains("ROLE_EDITOR"))
                {
                        returntrue;
                }
                returnfalse;
        }

        publicvoidsetRedirectStrategy(RedirectStrategyredirectStrategy)
        {
                this.redirectStrategy = redirectStrategy;
        }

        protectedRedirectStrategygetRedirectStrategy()
        {
                returnredirectStrategy;
        }

}
```

Above class overrides the handle method and redirects the user based on his role as follows –

- **Admin** will be redirected to **/admin**
- **Editor** will be redirected to **/editor**
- **User** will be redirected to **/home**

Spring Security Configuration for role-based login using successHandler is shown below.

```
@Override
protected void configure(final HttpSecurity http) throws Exception
{
    http
.authorizeRequests()
        // endpoints
.formLogin()
.loginPage("/login.html")
.loginProcessingUrl("/login")
.successHandler(customSuccessHandler())
        // other configuration
}
```

14.4 RESTRICT ACCESS BASED ON ROLES

Role-based access control (RBAC) is a mechanism that restricts system access. RBAC accomplishes this by assigning one or more "roles" to each user and giving each role different permissions. To protect sensitive data, mostly large-scale organizations use role-based access control to provide their employees with varying levels of access based on their roles and responsibilities.

The previous unit explained to secure the application using authentication and URL level security along with restrictions based on roles. The following sections explain to secure the application at service layer. Method-level security is used to restrict access based on Role. The last of this section explains to add security at view layer i.e.jsp.

Spring allows us to implement the security at the method level. Security applied at the method level restricts the unauthorized user to access the resource mapped by the method. How to enable method level security and restrict the method accessed based on role is explained below.

@EnableGlobalMethodSecurity

To Enable the method level security in spring we need to enable global method security as below:

```
@Configuration
@EnableGlobalMethodSecurity(
prePostEnabled = true,
securedEnabled = true,
  jsr250Enabled = true)
public class MethodSecurityConfig
  extends GlobalMethodSecurityConfiguration
{
}
```

- The prePostEnabled property enables Spring Security pre/post annotations
- The securedEnabled property determines if the @Secured annotation should be enabled
- The jsr250Enabled property allows us to use the @RoleAllowed annotation

@Secured

@Secured annotation is used to implement method level security based on Role. This annotation accepts list of allowed roles to be permitted. Hence, a user can access a method if he/she has been assigned at least one role into the list.

```
@Secured("ROLE_USER")
public String getUsername()
{
SecurityContextsecurityContext = SecurityContextHolder.getContext();
    return securityContext.getAuthentication().getName();
}
```

User who is having role as ROLE_USER can execute the above method.

```
@Secured({ "ROLE_USER", "ROLE_ADMIN" })
public booleanisValidUsername(String username)
{
    return userRoleRepository.isValidUsername(username);
}
```

A user having Either "ROLE_USER" OR "ROLE_ADMIN" can invoke above method.

@RolesAllowed Annotation

The @RolesAllowed annotation is the JSR-250's equivalent annotation of the @Secured annotation. Basically, we can use the @RolesAllowed annotation in a similar way as @Secured.

```
@RolesAllowed("ROLE_USER")
public String getUsername()
{
    //...
}


@RolesAllowed({ "ROLE_USER", "ROLE_ADMIN" })
public booleanisValidUsername(String username)
{
    //...
}
```

@PreAuthorize *and* @PostAuthorize Annotations

Both *@PreAuthorize* and *@PostAuthorize* annotations provide expression-based access control. The *@PreAuthorize* annotation checks the given expression before entering into the method, whereas, the *@PostAuthorize* annotation verifies it after the execution of the method and could alter the result.

```
@PreAuthorize("hasRole('ROLE_VIEWER')")
public String getUsername()
{
    //...
}
@PreAuthorize("hasRole('ROLE_USER') or hasRole('ROLE_ADMIN')")
public booleanisValidUsername(String username)
{
    //...
}
@PreAuthorize("#username == authentication.principal.username")
public String getMyRoles(String username)
{
    //...
}
```

getMyRoles() method can be invoked by a user only if the value of the argument *username* is the same as current principal's username

@PostAuthorize annotation is similar to @PreAuthorize so it can be replaced with @PostAuthorize.

Restriction at view layer

Access restriction based on role at view layer using spring security authorize tag is already explained. For details check the Section 14.2. Example is as follows-

```
<security:authorize access="!isAuthenticated()">
  Login
</security:authorize>
<security:authorize access="isAuthenticated()">
  Logout
</security:authorize>
```

Further we can also display certain information based on user role as-

```
<security:authorize access="hasRole('ADMIN')">
    Delete Users
</security:authorize>
```

☞Check Your Progress 2

1) Describe role-based login. Write the sample code to configure the role-based login.

   ……

   ……

   ……

   ……

   ……

2) What is Role Based Access Control (RBAC)? Write the sample configuration code to restrict the URL access based on role.

   ……

   ……

   ……

……

……

3) Explain @Secured and @RolesAllowed. What is the difference between them?

……

……

……

……

……

4) Which are all spring security annotations allowed to use SpEL?

……

……

……

……

……

5) What's the difference between @Secured and @PreAuthorize in spring security?

……

……

……

……

……

6) Explain spring security tag which restricts the access based on roles in jsp.

……

……

……

……

……

14.5 TESTING THE APPLICATION

This section writes a complete example with three types of users as Admin, Editor and User. UserType, Role and functionality has been defined in the table as follows:

User Type	Role	Functionality	Landing Page
Admin	ROLE_ADMIN, ROLE_EDITOR	Admin can create or edit a post/messages.	/admin
Editor	ROLE_EDITOR	Editor can only edit the post/messages.	/editor
User	ROLE_USER	User can only view the post/messages.	/home

In units 12 and 13, In-Memory authentication has been configured. Here JDBC authentication with mysql database has been explained. Tools and software required

to write the code easily, manage the spring dependencies and execute the application is as -

- Java 8 or above is required
- Maven
- Eclipse IDE
- Mysql
- Hibernate
- JPA

Directory Structure for Spring Project in eclipse is shown below. A few of the code is shown in Figure 14.1.

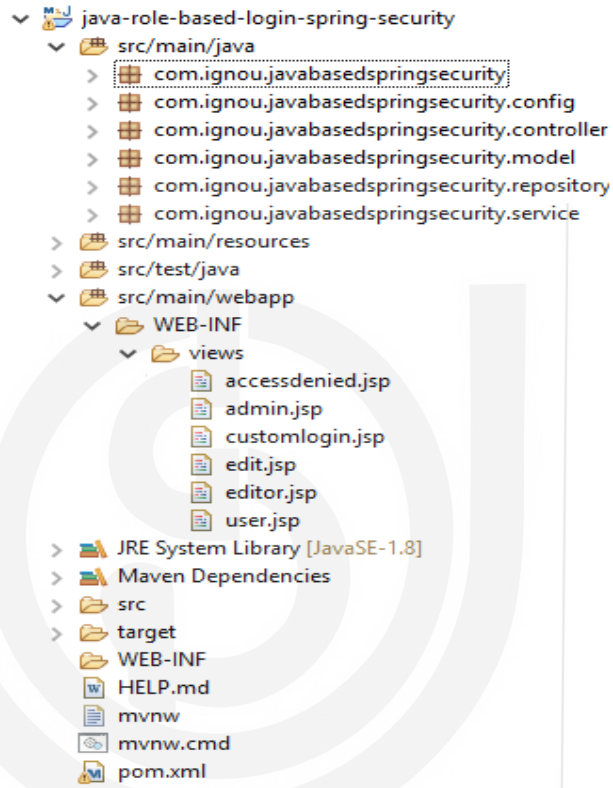

Figure 14.1: Project Folder Structure In Eclipse IDE

Spring Security Configuration classes in config packages – customSuccessHandler is already explained above for role-based login.

```
packagecom.ignou.javabasedspringsecurity.config;

@Component
publicclassCustomSuccessHandlerextendsSimpleUrlAuthenticationSuccessHandler
{
    privateRedirectStrategyredirectStrategy = newDefaultRedirectStrategy();

    @Override
    protectedvoidhandle(HttpServletRequestrequest, HttpServletResponseresponse,
    Authentication authentication)throwsIOException
    {
        String targetUrl = determineTargetUrl(authentication);

        if (response.isCommitted())
        {
            System.out.println("Can't redirect");
            return;
        }
```

```
        redirectStrategy.sendRedirect(request, response, targetUrl);
    }

    /*
     * This method extracts the roles of currently logged-in user and returns
     * appropriate URL according to his/her role.
     */
    protected String determineTargetUrl(Authentication authentication)
    {
        String url = "";
        Map<String, String>roleTargetUrlMap = new HashMap<>();
        roleTargetUrlMap.put("ROLE_USER", "/home");
        roleTargetUrlMap.put("ROLE_ADMIN", "/admin");
        roleTargetUrlMap.put("ROLE_Editor", "/editor");

        Collection<? extendsGrantedAuthority>authorities =
        authentication.getAuthorities();

        List<String>roles = newArrayList<String>();

        for (GrantedAuthoritya :authorities)
        {
                roles.add(a.getAuthority());
        }

        if (isAdmin(roles))
        {
                url = "/admin";
        }
        elseif (isEditor(roles))
        {
                url = "/editor";
        }
        elseif (isUser(roles))
        {
                url = "/home";
        }
        else
        {
                url = "/accessDenied";
        }

        returnurl;
    }

    privatebooleanisUser(List<String>roles)
    {
        if (roles.contains("ROLE_USER"))
        {
                returntrue;
        }
        returnfalse;
    }

    privatebooleanisAdmin(List<String>roles)
    {
        if (roles.contains("ROLE_ADMIN"))
        {
                returntrue;
        }
        returnfalse;
    }

    privatebooleanisEditor(List<String>roles)
    {
        if (roles.contains("ROLE_EDITOR"))
        {
                returntrue;
        }
        returnfalse;
    }

    publicvoidsetRedirectStrategy(RedirectStrategyredirectStrategy)
    {
        this.redirectStrategy = redirectStrategy;
    }
```

```
    protectedRedirectStrategygetRedirectStrategy()
    {
        returnredirectStrategy;
    }

}
```

SecurityConfig.java

Complete Security configuration with successHandler(customSuccessHandler) is as below-

```
packagecom.ignou.javabasedspringsecurity.config;

@EnableWebSecurity
publicclassSecurityConfigextendsWebSecurityConfigurerAdapter
{


@Autowired
    privateUserDetailsServicecustomUserService;

        @Autowired
    privateCustomSuccessHandlercustomSuccessHandler;

        @Autowired
        PasswordEncoderpasswordEncoder;

        @Bean
        publicPasswordEncoderpasswordEncoder()
        {
                returnnewBCryptPasswordEncoder();
        }

        @Override
        protectedvoidconfigure(AuthenticationManagerBuilderauth) throws Exception
        {
                auth
        .userDetailsService(customUserService)
        .passwordEncoder(passwordEncoder);
        }

        protectedvoidconfigure(HttpSecurityhttp) throws Exception
        {

                http.csrf().disable()
                .authorizeRequests()
                .antMatchers("/customlogin*").permitAll()
                .antMatchers("/","/home").hasRole("USER")
                .antMatchers("/editor/**").hasRole("EDITOR")
                .antMatchers("/admin/**").hasRole("ADMIN")
                .antMatchers("/edit/**").hasAnyRole("ADMIN","EDITOR")
                .antMatchers("/create/**").hasAnyRole("ADMIN")
                .anyRequest().authenticated()
                .and()
                .formLogin()
        .loginPage("/customlogin")
        .loginProcessingUrl("/signin")
        .successHandler(customSuccessHandler)
        .failureUrl("/customlogin?error=true")
                .and()
        .logout()
                .and().exceptionHandling().accessDeniedPage("/accessdenied");;
        }

}
```

Controller package Classes

LoginController.java

```
packagecom.ignou.javabasedspringsecurity.controller;

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

```
publicclassLoginController
{

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

}
```

HomeController.java

```
packagecom.ignou.javabasedspringsecurity.controller;

@Controller
publicclassHomeController
{

        privatestatic Map<Long, String>msgs = new HashMap<>();
        privatestaticlongmsgCount = 0;

        static
        {
                msgs.put(new Long(1), "Java Cryptography Architecture (JCA)");
                msgs.put(new Long(2), "Java Secure Socket Extension (JSSE)");
                msgCount = 2;
        }

        @RequestMapping(value="/home", method=RequestMethod.GET)
    public String user(Model model)
        {
                model.addAttribute("msgs", msgs);
         return"user";
        }

        @RequestMapping(value="/editor", method=RequestMethod.GET)
    public String editor()
        {
                return"editor";
        }

    @RequestMapping(value="/admin", method=RequestMethod.GET)
    public String admin()
        {

        return"admin";
        }

    @RequestMapping(value="/edit", method=RequestMethod.GET)
    public String edit(Model model)
        {
                model.addAttribute("msgs", msgs);
        return"edit";
        }

    @RequestMapping(value="/edit", method=RequestMethod.POST)
    public String editPost(@ModelAttributeMsgPostmsgPost, Model model)
        {
                longmsgId = msgPost.getId();
                if (msgs.get(msgId) != null)
                {
                msgs.put(msgPost.getId(), msgPost.getContent());
                }
                model.addAttribute("msgs",msgs);
                return"edit";
        }

    @RequestMapping(value="/create", method=RequestMethod.POST)
    publicStringcreatePost(@ModelAttributeMsgPostmsgPost, Model model)
        {
                msgCount += 1;
        msgs.put(msgCount, msgPost.getContent());
        model.addAttribute("msgs",msgs);
        return "edit";
        }
```

```
}
```

Service package classes

UserService.java

```
packagecom.ignou.javabasedspringsecurity.service;

@Service
@Transactional
publicclassUserServiceimplementsUserDetailsService
{

  @Autowired
  privateUserRepositoryuserRepository;

  @Override
  publicUserDetailsloadUserByUsername(String email)
  throwsUsernameNotFoundException
  {

    User user = userRepository.findByEmail(email).orElseThrow(()-
    >newUsernameNotFoundException(email + "not found"));
    returnneworg.springframework.security.core.userdetails.User(user.getEmail(),
    user.getPassword(),getAuthorities(user));
  }

  privatestatic Collection<? extendsGrantedAuthority>getAuthorities(User user)
  {
    String[] userRoles = user.getRoles().stream().map((role) ->
    role.getName()).toArray(String[]::new);
    Stream.of(userRoles).forEach(System.out::print);
    Collection<GrantedAuthority>authorities =
    AuthorityUtils.createAuthorityList(userRoles);
    returnauthorities;
  }

}
```

Jsp in WEB-INF/views

Note: customlogin.jsp has been used same as explained in Unit 13.

admin.jsp

```
<%@taglibуri="http://www.springframework.org/security/tags"prefix="security"%>
<%@pagelanguage="java"contentType="text/html; charset=ISO-8859-1"pageEncoding="ISO-
8859-1"%>

<html>
<head>
<metahttp-equiv="Content-Type"content="text/html; charset=ISO-8859-1">
<title>Admin page</title>
</head>
<body>
    Dear <strong><security:authenticationproperty="principal.username"/></strong>,
Welcome to Admin Page.
<p>You are having roles as
<strong><security:authenticationproperty="principal.authorities"/></strong>
<p><ahref="/edit">Edit</a>
<p><ahref="/logout">Logout</a>
</body>
</html>
```

editor.jsp

```
<%@taglibуri="http://www.springframework.org/security/tags"prefix="security"%>
<%@pagelanguage="java"contentType="text/html; charset=ISO-8859-1"pageEncoding="ISO-
8859-1"%>

<html>
<head>
<metahttp-equiv="Content-Type"content="text/html; charset=ISO-8859-1">
<title>Editor page</title>
</head>
```

```
<body>
    Dear <strong><security:authenticationproperty="principal.username"/></strong>,
Welcome to Editor Page.
<p>You are having roles as
<strong><security:authenticationproperty="principal.authorities"/></strong>
<p><ahref="/edit">Edit</a>
<p><ahref="/Logout">Logout</a>
</body>
</html>
```

user.jsp

```
<%@taglibуri="http://www.springframework.org/security/tags"
        prefix="security"%>
<%@taglibprefix="c"uri="http://java.sun.com/jsp/jstl/core"%>
<%@pagelanguage="java"contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>

<html>
<head>
<metahttp-equiv="Content-Type"content="text/html; charset=ISO-8859-1">
<title>Welcome page</title>
<style>
table.border,th.border,td.border
{
        margin-left: auto;
        margin-right: auto;
        border: 1px solid black;
}
</style>
</head>
<body>
        Dear
        <strong><security:authenticationproperty="principal.username"/></strong>,
        Welcome to Home Page.
        <p>
                You are having roles as <strong><security:authentication
                                property="principal.authorities"/></strong>
        <tableclass="border">
                <tr>
                        <thclass="border">Msg Id</th>
                        <thclass="border">Message</th>
                </tr>
                <c:forEachitems="${msgs}"var="msg">
                        <tr>
                                <tdclass="border">${msg.key}</td>
                                <tdclass="border">${msg.value}</td>
                        </tr>
                </c:forEach>
        </table>
        <p>
                <ahref="/Logout">Logout</a>
</body>
</html>
```

Execute the project either from eclipse or command line. After successful execution of the application, security configured application can be accessed using browser on localhost:8080

The following users are configured for the testing of role-based login example.

- Admin – UserName: **admin@gmail.com**Paasword: **admin**
- Editor – UserName: **editor@gmail.com**Paasword: **editor**
- User – UserName: **user@gmail.com**Paasword: **user**

Execution Result

If a user tries to access **localhost:8080** and the user is not logged in, spring will redirect to the user on **localhost:8080/customlogin** as shown in Figure 14.2.

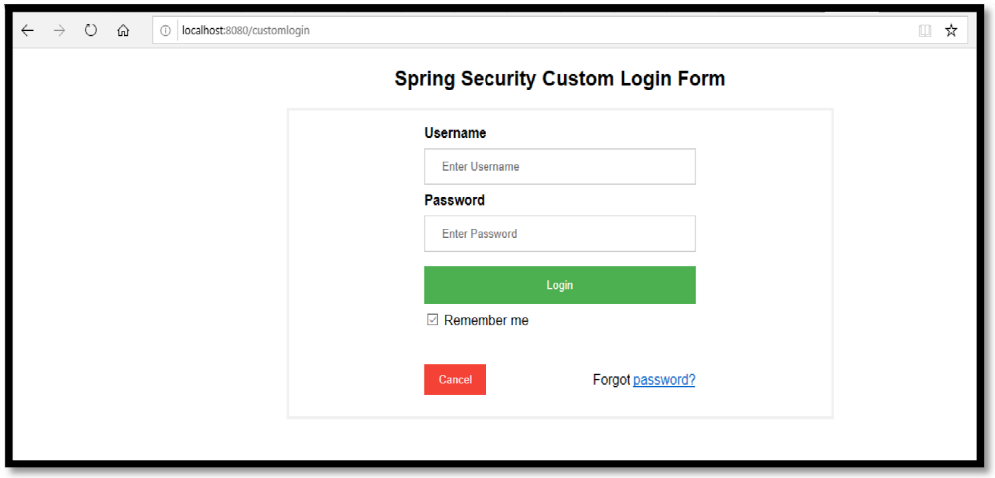

Figure 14.2: Spring Security Custom Login Page

Admin Login (Shown in Figure 14.3.

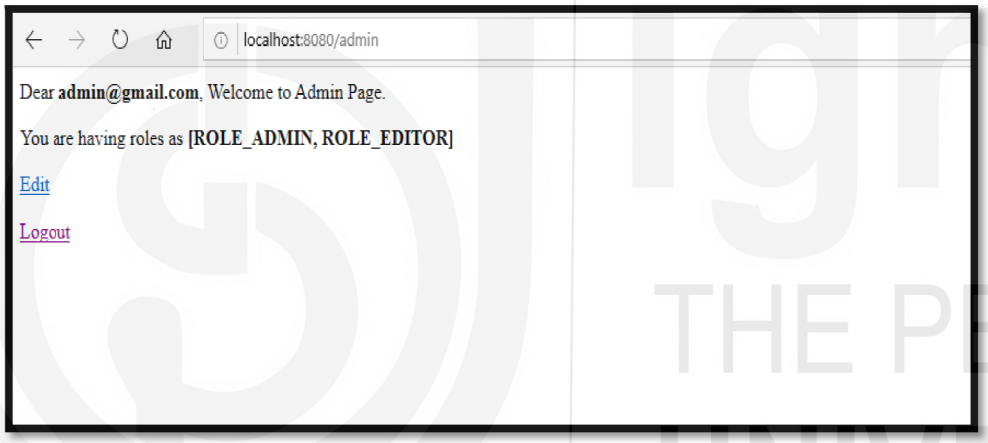

Figure14.3: Admin User Home Screen

On click on Edit button admin will see the below screen (Figure 14.4). Admin has only permission to create new messages as explained the feature of each user in the beginning of this section.

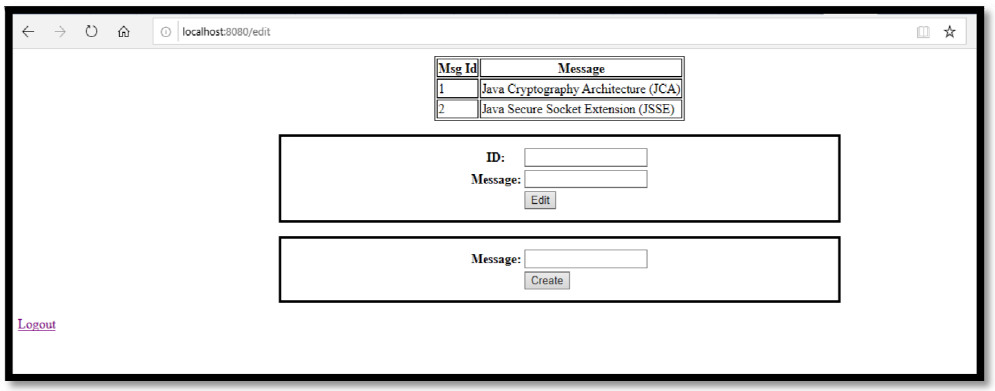

Figure14.4: Edit and Create Screen For Admin User

When Editor logs in, you will get the below screen (Figure 14.5).

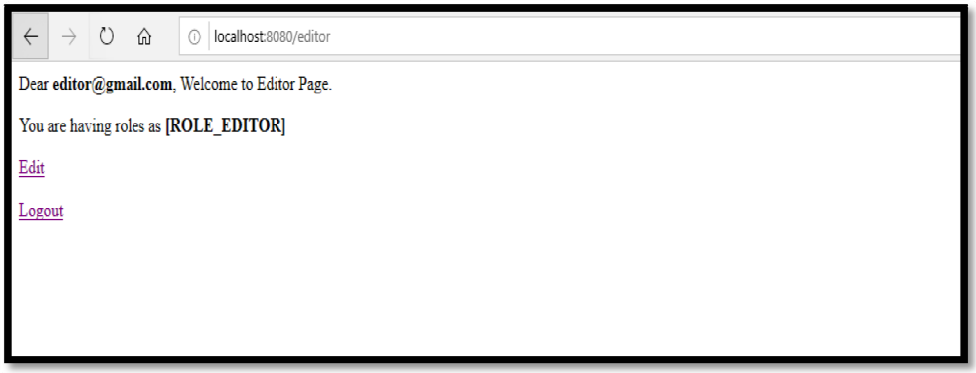

Figure 14.5: Editor Role User Home Screen

On click on Edit button, editor can only edit the already existing messages. Check edit.jsp file in which **<security:authorize access="hasRole('ADMIN')">**security tag has been used to allow only Admin user to create new messages (Figure 14.6).

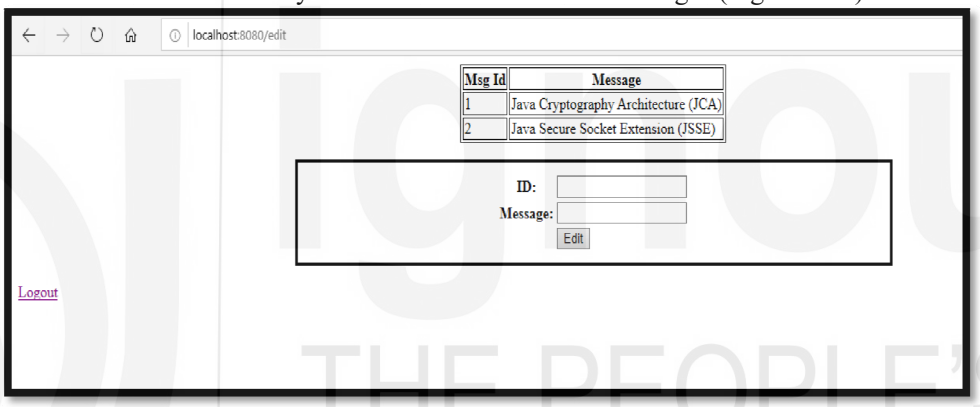

Figure14.6: Edit Screen from Editor Role User

When a user logs in, he/she will get the below screen(Figure 14.7) as you can see that normal user does not have any option to edit the message.

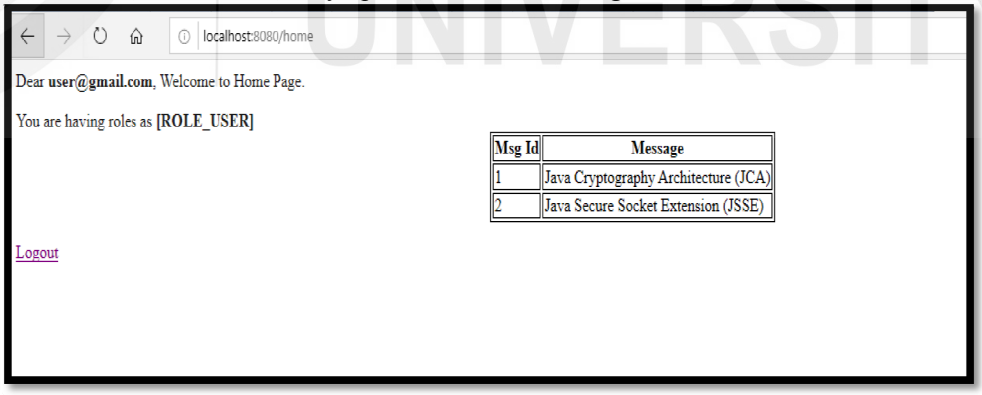

Figure14.7: General User Home Screen

14.5.1 Unit Tests in Spring Boot

Production-ready code should have well-written Unit test cases as well as Integration Test. Good unit test cases should have code coverage of 80% or more. Writing good unit test cases is an art, and sometimes it's difficult to write good code coverage unit test cases. Spring provides an efficient way to write Unit Test cases as well Integration Test cases. In Unit 13, various annotations have been described for Integration Testing

with examples. @SpringBootTest is used for integration testing, and it is useful when we need to bootstrap the entire container. This section will describe the best practices to write unit test cases as well as available annotations will be described.

@SpringBootTest

```
@SpringBootTest

class StudentEnrollmentUseCaseTest

{


 @Autowired

 private EnrollmentServiceenrollmentService;


 @Test

 void studentEnrollemnt()

 {

  Student student = new Student("Jack", "jack@mail.com");

  Student enrolledStudent = enrollmentService.enroll(student);

assertThat(enrolledStudent.getEnrollmentNo()).isNotNull();

 }

}
```

A good test case takes only few milliseconds, but the above code might take 3-4 seconds. The above test case execution takes only a few milliseconds, and the rest of the 4 seconds is being taken by @SpringBootTest to set up the complete Spring Boot Application Context. In the above test case complete Spring Boot Application Context has been set up to autowire the EnrollmentService instance into the test case. Unit test cases set up time can be reduced a lot with the use of Mock instances and appropriate annotation instead of @SpringBootTest.

A well-structured web application consists of multiple layer such as Controllers, Services and Repository. Controllers are responsible to handle the client request forwarded by dispatcherServlet. The controller uses Service classes for business logic and Service classes uses Repository classes to interact with databases. Each layer can be tested independently with Unit test cases. The following section will describe the important annotations available in Spring to write the good unit test with a few milliseconds of execution time.

14.5.1.1 Data Layer or Repository Layer testing with @DataJpaTest

An entity class named as Student which is having properties as id and name

```
@Entity
@Table(name = "student")
public class Student
{
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Size(min = 3, max = 20)
    private String name;

    // constructors, getters and setters

}
```

StudentRepository with Spring Data Jpa

```
@Repository
public interface StudentRepository extends JpaRepository<Student, Long>
{

    public Student findById(Long id);
     public List<Student>findByName(String name);

}
```

Persistence layer unit test cases code set up shown below

```
@DataJpaTest
public class UserRepositoryTest

{

    @Autowired
    private TestEntityManagerentityManager;

    @Autowired
    private UserRepositoryuserRepository;
    // test cases

}
```

@DataJpaTest focuses only on JPA components needed for testing the persistence layer. Instead of initializing the complete Spring Boot Application Context, as in the case of @SpringBootTest, initializes only the required configuration relevant to JPA tests. Thus, Unit test case set-up time is very less. @DataJpaTest does the following configurations.\

- setting Hibernate, Spring Data, and the DataSource
- configuring H2, an in-memory database
- turning on SQL logging
- performing an @EntityScan

A complete example of persistence layer unit test case using @DataJpaTest

```
@DataJpaTest
publicclassUserRepositoryTest
{

    @Autowired
privateTestEntityManagerentityManager;

@Autowired
privateUserRepositoryuserRepository;

@BeforeEach
privatevoidsetUp()
    {
        User user = newUser();
user.setEmail("testadmin@gmail.com");
user.setName("Admin");
user.setPassword("admin@123");
entityManager.persist(user);
entityManager.flush();
    }
```

```
@Test
publicvoidfindByEmailTest()
    {

        String email = "testadmin@gmail.com";

// Call userRepository to get user record
        Optional<User>found = userRepository.findByEmail(email);
        String getUserEmail = found.get().getEmail();

assertThat(getUserEmail).isEqualTo(email);
    }

}
```

14.5.1.2 Service Layer and Controller layer testing with@MockBean

In MVC project, Controllers are dependent on Services, and Services are dependent on Repositories. However, Controllers and Services should be testable without knowing the complete implementation of dependencies.
@MockBean can be used to mock the required dependency. Spring boot @MockBean annotation used to add mocks to a Spring ApplicationContext.

A complete example of Service layer unit test case with @MockBean

```
@ExtendWith(SpringExtension.class)
publicclassUserServiceTest
{
  @TestConfiguration
  staticclassUserServiceTestConfiguration
  {

    @Bean
    publicUserServiceuserService()
    {
      returnnewUserService();
    }
  }
  @MockBean
  privateUserRepositoryuserRepository;
  @Autowired
  privateUserServiceuserService;
  @BeforeEach
  publicvoidsetUp()
  {
    User user = newUser();
    List<Role>roleList = newArrayList<>();

    Role adminRole = newRole();
    adminRole.setName("Admin");
    Role editorRole = newRole();
    editorRole.setName("Editor");
    roleList.add(adminRole);
    roleList.add(editorRole);

    user.setEmail("testadmin@gmail.com");
    user.setName("Admin");
    user.setPassword("admin@123");
    user.setRoles(roleList);

    Mockito.when(userRepository.findByEmail(user.getEmail())).thenReturn(Optional.of(u
ser));
  }

  @Test
  publicvoidwhenValidName_thenUserShouldBeFoundTest()
  {
    String email = "testadmin@gmail.com";
    UserDetailsfound = userService.loadUserByUsername(email);
    assertThat(found.getUsername()).isEqualTo(email);
  }
}
```

All unit test cases can be executed by right click on the project -> Run As -> Junit Test as shown below screenshot (Figure 14.8).

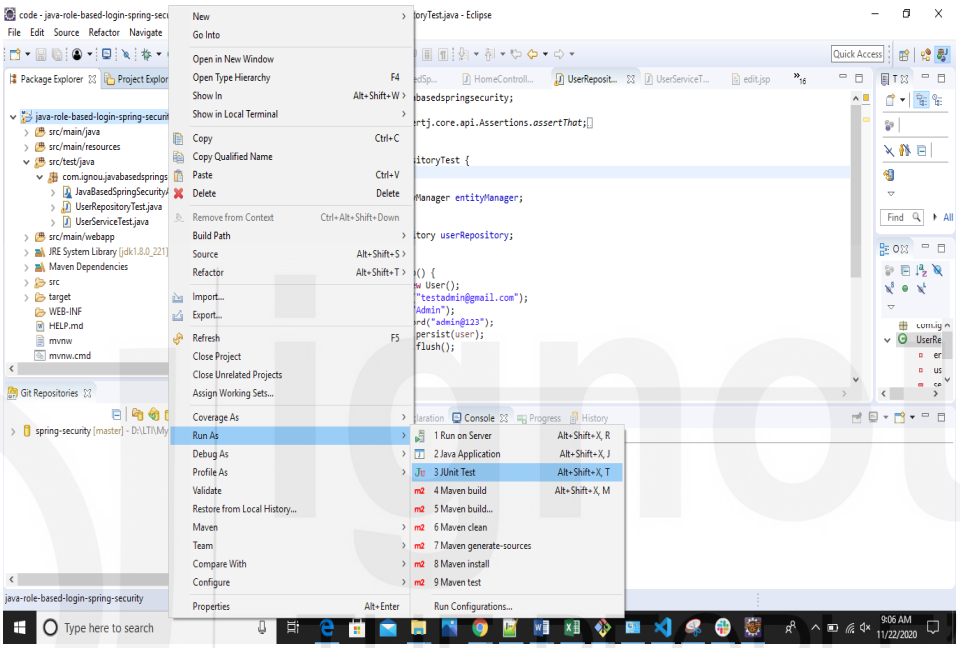

Figure 14.8: Unit Test Case Execution From Eclipse IDE

Successful execution of Unit Test case will output the following (Figure 14.9). As you can see all test cases are passed,which is shown by the green tick and green progress bar.

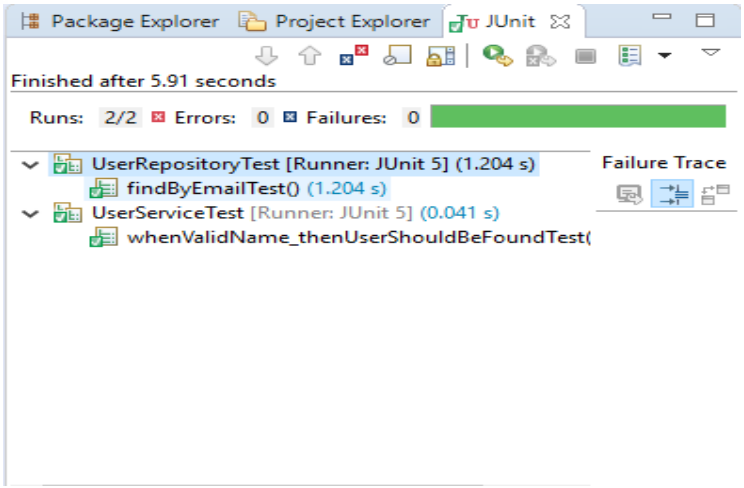

Figure14.9: Unit Test Cases Execution Status

14.6 CROSS SITE REQUEST FORGERY (CSRF)

In Unit 12, section 12.3 CSRF concept and ways of its execution have been described.CSRF is a type of attack in which attackers trick a victim into making a request that utilizes their authentication or authorization. The victim's level of permission decides the impact of CSRF attack. Execution of CSRF attack consists of mainly two-part.

- Trick the victim into clicking a link or loading a web page
- Sending a crafted, legitimate-looking request from victim's browser to website

For a better understanding of CSRF, let us have a look at a concrete example of bank money transfer.
Suppose that bank's website provides a form to transfer money from the currently logged in user to another bank account.
For example, the HTTP request might look like:

```
POST /transfer HTTP/1.1
Host: bank.example.com
Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly
Content-Type: application/x-www-form-urlencoded

amount=100.00&account=1234
```

Now consider that user authenticate to bank's website and then, without logging out, visit an evil website. The evil website contains an HTML page with the following form:

```
<form action="https://bank.example.com/transfer" method="post">
<input type="hidden"
         name="amount"
         value="100.00"/>
<input type="hidden"
         name="account"
         value="evilsAccountNumber"/>
<input type="submit"
         value="Win Money!"/>
</form>
```

User like to win money, and he clicks on the Win Money! Button. As user clicks on button, unintentionally he transferred $100 to a malicious user. While evil website can not see your cookies, but the browser sent the cookie associated with bank along with the request.
The whole process can be automated using JavaScript, and onPage load script can be executed to perform CSRF. The most popular method to prevent Cross-site Request Forgery is to use a randomly generated token that is associated with a particular user and that is sent as a hidden value in every state-changing form in the web app. This token, called an anti-CSRF token (often abbreviated as CSRF token) or a synchronizer token. When a request is submitted, the server must look up the expected value for the parameter and compare it against the actual value in the request. If the values do not match, the request should fail.

Spring Security CSRF Protection

Necessary steps to use Spring Security's CSRF protection are as follows-

- Use proper HTTP verb
- Configure CSRF protection

- Include the CSRF token

Use Proper HTTP Verb

The use of proper HTTP verb plays a vital role to protect a website against CSRF attack. We need to be assured that the application is using PATCH, POST, PUT, and/or DELETE for anything that modifies state. This is not a limitation of Spring Security's support, but instead a general requirement for proper CSRF prevention.

Configure CSRF protection

By default, CSRF protection is enabled in Spring java configuration. It can be disabled as below:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter
{
        @Override
        protected void configure(HttpSecurity http) throws Exception
        {
             http
             .csrf().disable();
        }
}
```

Include the CSRF Token

The last step is to include the CSRF token all PATCH, POST, PUT, and DELETE methods. Csrfinput Tag is described into section 14.2 and used it in edit.jsp in Section 14.5. In edit.jsp we used two approach to include csrf token as follows-

- ***<input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}" />***
- ***<security:csrfInput/>***

Following example will give you experimental and real feel of CSRF attack in the application executed in Section 14.5. Steps to perform CSRF attack simulation are as below-

Step 1: To simulate the evil website create a html file with below content. You can give it any name. Let us assume that filename is as **evil.html** and save this file to any location.

```
<html>
<body>
<form action="http://localhost:8080/edit" method="POST" id="attack">
         <input type="hidden" name="id" value="2">
         <input type="hidden" name="content" value="Hackers Msg">
         <input type="button"
onclick="document.getElementById('attack').submit()" value="Click me to Won
Prize"></input>
</form>
</body>
</html>
```

Step 2: Start the application tested into Section 14.5 and login as Admin or Editor user. Here consideration is that Admin user do login into the system (Figure 14.10).

Figure 14.10: Admin Role User Home Screen

Step 3: For CSRF attack to place User must be logged in to the system. In step 2, Admin has logged in. Now open evil.html in the same browser in which admin has logged and click on the button. Once you click the button, the following screen (Figure 14.11) will appear with Hacker msg.

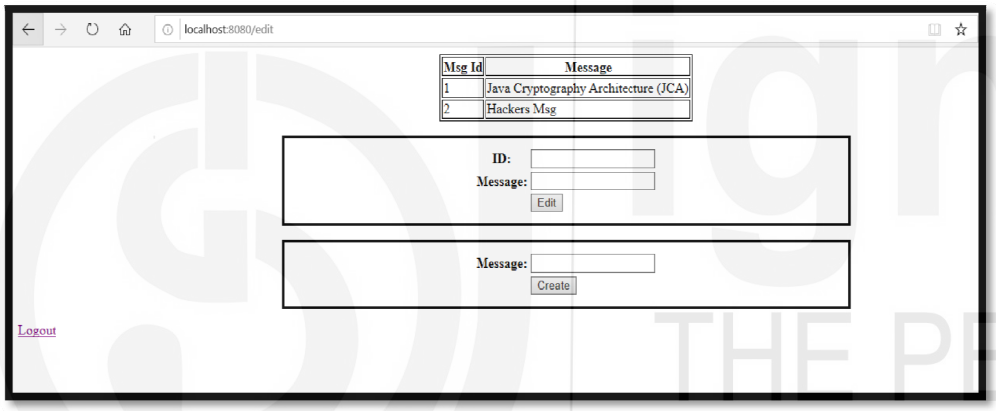

Figure 14.11: CSRF Attack

As you can see in above screen that malicious user has edited the msg. Close the evil.html. From admin login, click on edit link, and there also you will find the modified msg by malicious user.

Now we will make the required changes to protect the website against CSRF attacks. We have configured security in Security.java file. You can check that csrf is disabled there. You can just remove csrf.disable() and automatically csrf protection will be enabled. We have already included token in jsp so no need to make any changes in jsp. Perform all the steps again, and this time in step 3 you will get the below screen (Figure 14.12) .

Figure 14.12: Website Protected Against CSRF Attack

As you can see now,the edit is being denied since we have enabled CSRF protection and the malicious user is unable to edit or create the msg.

You can try to create a new message as a malicious user by disabling csrf protection. Make the changes in evil.html.

☛Check Your Progress 3

1) Explain @SpringBootTest. Why is it not suitable for unit test cases?

……

……

……

……

……

2) Describe @MockBean with an example.

……

……

……

……

……

3) What is Cross Site Request Forgery (CSRF)? What is the necessary condition for CSRF to take place?

……

……

……

……

……

4) How do you make the Spring Application secured against CSRF attack? Discuss the necessary steps along with sample code to enable CSRF in Spring.

……

……

……

……

……

14.7 SUMMARY

In the Unit, you learnt various core components of Spring Security such as SecurityContextHolder, SecurityContext, Authentication, Principal etc. Next you

learnt how to get logged in user details in controllers using Authentication and Principal and many more.

Gradually you learnt various Spring Security tags such as authorize, authentication, csrfInput etc. to secure the view layer jsp and display the user details on jsp using these tags.

Next you learnt about role-based login. In role-based login you learnt to redirect users on different URL based on assigned role of the User. You learnt about AuthenticationSuccessHandler and configuration using successHandler().

This Unit has described how to secure the service layer methods using various annotations such as @Secured, @RoleAllowed, @PreAuthorize, @PostAuthorize etc.

You executed a complete application in which you learnt how to implement the above concept programmatically. In the lastof the unit, you learnt how to enable CSRF protection in Spring application. You got the real feel of CSRF attack with executed example, and this will enable you to understand CSRF attack easily.

14.8 SOLUTIONS/ ANSWER TO CHECK YOUR PROGRESS

☛Check Your Progress 1

1) All Authentication implementations in Spring is having a list of GrantedAuthority objects. These represent the authorities that have been granted to the principal. AuthenticationManager inserts all the GrantedAuthority objects into the Authentication Object. Hierarichal Structure is as below (Figure 14.13).

Figure 14.13: Authorities in Authentication object

2) UserName using HttpServletRequest can be accessed as follows-

```
@Controller
public class UserSecurityController
{

    @RequestMapping(value = "/username", method = RequestMethod.GET)
    @ResponseBody
    public String loggeInUserName(HttpServletRequest request)
    {
        Principal principal = request.getUserPrincipal();
        return principal.getName();
    }
}
```

3) Authentication tag can be used get logged in user details and authorities on jsp as below-

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

For details refer authentication tag in the Section 14.2

☛Check Your Progress 2

1) **AuthenticationSuccessHandler**is an Interface, and Spring provides SimpleUrlAuthenticationSuccessHandler, which implement this interface. To create the custom success handler we can extend the SimpleUrlAuthenticationSuccessHandler class and can override the handle() method to provide the required logic to for redirect URL. For role based login, you need to configure **successHandler(AuthenticationSuccessHandlersuccessHandler)** instead of ***defaultSuccessUrl(Sring s).*** Configuration for custom success handler is as below:

```
@Override
protected void configure(final HttpSecurity http) throws Exception
{
    http
.authorizeRequests()
        // endpoints
.formLogin()
.loginPage("/login.html")
.loginProcessingUrl("/login")
.successHandler(customSuccessHandler())
        // other configuration
}
```

2) For RBAC refer section 14.4. Configuration to restrict the URL access based on role is as below-

```
protectedvoidconfigure(HttpSecurityhttp) throws Exception
{

        http.csrf().disable()
        .authorizeRequests()
        .antMatchers("/customlogin*").permitAll()
        .antMatchers("/","/home").hasRole("USER")
        .antMatchers("/editor/**").hasRole("EDITOR")
        .antMatchers("/admin/**").hasRole("ADMIN")
        .antMatchers("/edit/**").hasAnyRole("ADMIN","EDITOR")
        .antMatchers("/create/**").hasAnyRole("ADMIN")
        .anyRequest().authenticated()
        .and()
        .formLogin()
        .loginPage("/customlogin")
        .loginProcessingUrl("/signin")
.successHandler(customSuccessHandler)
.failureUrl("/customlogin?error=true")
        .and()
.logout()
        .and().exceptionHandling().accessDeniedPage("/accessdenied");;
}
```

In above configuration
URL pattern "/" and "/home" is accessible only to user which is having role as ROLE_USER.
URL pattern "/editor/**" is accessible only to user having role as ROLE_EDITOR
URL pattern "/admin/**" is accessible only to user having role as ROLE_ADMIN

URL pattern "/edit/**" is accessible to user having role as ROLE_ADMIN or ROLE_EDITOR
URL pattern "/create/**" is permissible only to user having role as ROLE_ADMIN

3) @Secured and @RolesAllowed both annotations provide method-level security into Spring Beans. @Secured is Spring Security annotation from version 2.0 onwards Spring Security. But @RolesAllowed is JSR 250 annotation. Spring Security provides the support for JSR 250 annotation as well for method level security. @RolesAllowed provides role-based security only.

4) @PreAuthorize and @PostAuthorize allow to use SpEL. These annotations support expression attributes to allow pre and post-invocation authorization checks.

5) If you wanted to do something like access the method only if the user has Role1 and Role2 the you would have to use @PreAuthorize @PreAuthorize("hasRole('ROLE_role1') and hasRole('ROLE_role2')") Using @Secured({"role1", "role2"}) is treated as an OR

6) For details check the Section 14.2. Example is as follows-

```
<security:authorize access="!isAuthenticated()">
  Login
</security:authorize>
<security:authorize access="isAuthenticated()">
  Logout
</security:authorize>
```

Further we can also display certain information based on user role as-

```
<security:authorize access="hasRole('ADMIN')">
    Delete Users
</security:authorize>
```

☞Check Your Progress 3

1) @SpringBootTest is used for integration testing and it is useful when we need to bootstrap the entire container. The @SpringBootTest annotation tells Spring Boot to look for a main configuration class (one with @SpringBootApplication, for instance) and use that to start a Spring application context. @SpringBootTest by default starts searching, a class annotated with @SpringBootConfiguration, in the current package of the test class and then searches upwards through the package structure. It reads the configuration from @SpringBootConfiguration to create an application context. This class is usually our main application class since the @SpringBootApplication annotation includes the @SpringBootConfiguration annotation. It then creates an application context very similar to the one that would be started in a production environment.

2) In the MVC project, Controllers are dependent on Services, and Services are dependent on Repositories. However, Controllers and Services should be testable without knowing the complete implementation of dependencies. @MockBean can be used to mock the required dependency. Spring boot @MockBean annotation used to add mocks to a Spring ApplicationContext.
   - @MockBean allows to mock a class or an interface.

- @MockBean can be used on fields in either @Configuration classes or test classes that are @RunWith the SpringRunner as well as a class level annotation.
  @MockBean at field level

```
@RunWith(SpringRunner.class)
public class LoginControllerTest
{

   @MockBean
   private LoginServiceloginService;

}
```

  @MockBean at class level

```
@RunWith(SpringRunner.class)
@MockBean(LoginService.class)
public class LoginControllerTest
{

   @Autowired
   private LoginServiceloginservice;

}
```

- Mocks can be registered by type or by bean name.
- Any existing single bean of the same type defined in the context will be replaced by the mock. If no existing bean is defined a new one will be added.

3) For CSRF details, refer Unit 12 section 12.3 and Unit 14 section 14.6. Necessary condition for CSRF attack to succeed is that that the victim must be logged in and attacker is able to trick the victim into clicking a link or loading a web page is done through social engineering or using a malicious link.

4) CSRF is a type of attack in which attackers trick a victim into making a request that utilizes their authentication or authorization. The victim's level of permission decides the impact of CSRF attack. Execution of CSRF attack consists of mainly two-part.
   - Trick the victim into clicking a link or loading a web page
   - Sending a crafted, legitimate-looking request from victim's browser to the website

   Necessary steps to use Spring Security's CSRF protection are as follows-
   - Use proper HTTP verb
   - Configure CSRF protection
   - Include the CSRF token

14.9 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, "**J**ava 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)
- Tomcy John, "Hands-On Spring Security 5 for Reactive Applications",Packt Publishing,2018
- https://www.baeldung.com/spring_redirect_after_login
- https://digitalguardian.com/blog/what-role-based-access-control-rbac-examples-benefits-and-more
- https://docs.spring.io/spring-security/site/docs/5.0.x/reference/html/csrf.html
- https://docs.spring.io/spring-framework/docs/current/reference/html/testing.html
- https://www.baeldung.com/spring-security-csrf
- https://www.baeldung.com/java-spring-mockito-mock-mockbean