COMP S380F Web Applications: Design and Development
Lab 11: Spring Security

We will have exercises on using Spring Security to authenticate users. The topics below will be covered:

- Creating a custom user service for Spring Security
- User management
- Adding custom login, logout, remember-me
- Securing web pages and JSP contents

Task 1: Creating user and role repositories & custom user service using Spring Data JPA

Download the branch "**lab11**" of the Github repository "**https://github.com/cskeith/380_2024.git**".

1. In **`build.gradle`**, add dependencies for Spring Security. Reload the project in the Gradle window.

```
implementation 'org.springframework.boot:spring-boot-starter-security'
implementation 'org.springframework.security:spring-security-taglibs'
```

2. As in Lab 10, we reset the H2 database using two SQL files whenever we start the web application. In **`resources/sql/schema.sql`**, add the CREATE TABLE statements for the user store.

```
DROP TABLE IF EXISTS user_roles;
DROP TABLE IF EXISTS users;
CREATE TABLE users (
    username VARCHAR(50) NOT NULL,
    password VARCHAR(50) NOT NULL,
    PRIMARY KEY (username)
);
CREATE TABLE IF NOT EXISTS user_roles (
    user_role_id INTEGER GENERATED ALWAYS AS IDENTITY,
    username VARCHAR(50) NOT NULL,
    role VARCHAR(50) NOT NULL,
    PRIMARY KEY (user_role_id),
    FOREIGN KEY (username) REFERENCES users(username)
);
```

3. In **`resources/sql/data.sql`**, add the INSERT statements for three user accounts:
   - `keith`: both a user and an admin.
   - `john`: an admin.
   - `mary`: a user.

   To indicate that the passwords of the users are stored as plain texts in the database, we need to add the prefix `{noop}` to each password so as to specify that the `NoOpPasswordEncoder` is used.

```
INSERT INTO users VALUES ('keith', '{noop}keithpw');
INSERT INTO user_roles(username, role) VALUES ('keith', 'ROLE_USER');
INSERT INTO user_roles(username, role) VALUES ('keith', 'ROLE_ADMIN');

INSERT INTO users VALUES ('john', '{noop}johnpw');
INSERT INTO user_roles(username, role) VALUES ('john', 'ROLE_ADMIN');

INSERT INTO users VALUES ('mary', '{noop}marypw');
INSERT INTO user_roles(username, role) VALUES ('mary', 'ROLE_USER');
```

4. One user may have many roles, so the table `users` has a one-to-many relationship with the table `user_roles`. In package **`hkmu.comps380f.model`**, create two entity classes **`TicketUser`** and **`UserRole`** for the two tables `users` and `user_roles`.

   Note that an entity class requires a **default constructor**, and all ambiguous annotations are from **`jakarta.persistence`**. Eager fetching of user information is required by ***Spring Security***.

```
@Entity
@Table(name = "users")
public class TicketUser {
    @Id
    private String username;

    private String password;

    @OneToMany(mappedBy = "user", fetch = FetchType.EAGER,
            cascade = CascadeType.ALL, orphanRemoval = true)
    private List<UserRole> roles = new ArrayList<>();

    public TicketUser() {}

    public TicketUser(String username, String password, String[] roles) {
        this.username = username;
        this.password = "{noop}" + password;
        for (String role : roles) {
            this.roles.add(new UserRole(this, role));
        }
    }

    // getters and setters of all properties
}
```

```
@Entity
@Table(name = "user_roles")
public class UserRole {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "user_role_id")
    private int id;

    @Column(insertable = false, updatable = false)
    private String username;

    private String role;

    @ManyToOne
    @JoinColumn(name = "username")
    private TicketUser user;

    public UserRole() {
    }

    public UserRole(TicketUser user, String role) {
        this.user = user;
        this.role = role;
    }

    // getters and setters of all properties
}
```

5. With the entity classes, we add the repositories for `TicketUser` and `UserRole` to the **dao** package `hkmu.comps380f.dao`:

```
public interface TicketUserRepository extends JpaRepository<TicketUser, String> {
}
```

```
public interface UserRoleRepository extends JpaRepository<UserRole, Integer> {
}
```

6. We apply the **Controller-Service-Repository** design pattern, and define a custom Spring Security user service in the **dao** package `hkmu.comps380f.dao`:

```
@Service
public class TicketUserService implements UserDetailsService {
    @Resource
    TicketUserRepository ticketUserRepo;

    @Override
    public UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException {
        TicketUser ticketUser = ticketUserRepo.findById(username).orElse(null);
        if (ticketUser == null) {
            throw new UsernameNotFoundException("User '" + username + "' not found.");
        }
        List<GrantedAuthority> authorities = new ArrayList<>();
        for (UserRole role : ticketUser.getRoles()) {
            authorities.add(new SimpleGrantedAuthority(role.getRole()));
        }
        return new User(ticketUser.getUsername(), ticketUser.getPassword(), authorities);
    }
}
```

Spring Security uses the method **`loadUserByUsername`** to obtain the information of a user. This method throws an exception **`UsernameNotFoundException`** if the specified user does not exist. If the specified user exists, we will return a ***Spring Security***'s **`User`** object. The user roles are represented as a list of **`GrantedAuthority`** objects.

7. In each of the JSP pages, add the following logout button in the beginning of `<body>`:

```
<c:url var="logoutUrl" value="/logout"/>
<form action="${logoutUrl}" method="post">
    <input type="submit" value="Log out" />
    <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
</form>
```

8. We don't need the username when creating a ticket. Update **`TicketController`**:
   - The username can be obtained using the **`Principal`** object, as follows:

```
@PostMapping("/create")
public View create(Form form, Principal principal) throws IOException {
    long ticketId = tService.createTicket(principal.getName(),
            form.getSubject(), form.getBody(), form.getAttachments());
    return new RedirectView("/ticket/view/" + ticketId, true);
}
```

   - For the **`Form`** object, remove the property **`customerName`** and its **getter** and **setter**.
   - In **`add.jsp`**, we remove the **`<form:label>`** and **`<form:input>`** for **`customerName`**.

9. Run the web application. You can login using any user account. Note that in each HTML form created by **`<form:form>`**, Spring will automatically add the hidden field "**`_csrf`**"

Task 2: User management

1. Create the following **UserManagementService** in the **dao** package hkmu.comps380f.dao:

```
@Service
public class UserManagementService {
    @Resource
    private TicketUserRepository tuRepo;

    @Transactional
    public List<TicketUser> getTicketUsers() {
        return tuRepo.findAll();
    }

    @Transactional
    public void delete(String username) {
        TicketUser ticketUser = tuRepo.findById(username).orElse(null);
        if (ticketUser == null) {
            throw new UsernameNotFoundException("User '" + username + "' not found.");
        }
        tuRepo.delete(ticketUser);
    }

    @Transactional
    public void createTicketUser(String username, String password, String[] roles) {
        TicketUser user = new TicketUser(username, password, roles);
        tuRepo.save(user);
    }
}
```

Note that **UsernameNotFoundException** is also an **RuntimeException**, which will be automatically handled by `@Transactional` .

2. Create the following controller in the **controller** package hkmu.comps380f.controller:

```
@Controller
@RequestMapping("/user")
public class UserManagementController {
    @Resource
    UserManagementService umService;

    @GetMapping({"", "/", "/list"})
    public String list(ModelMap model) {
        model.addAttribute("ticketUsers", umService.getTicketUsers());
        return "listUser";
    }

    public static class Form {
        private String username;
        private String password;
        private String[] roles;

        // getters and setters for all properties
    }

    @GetMapping("/create")
    public ModelAndView create() {
        return new ModelAndView("addUser", "ticketUser", new Form());
    }

    @PostMapping("/create")
    public String create(Form form) throws IOException {
        umService.createTicketUser(form.getUsername(),
                form.getPassword(), form.getRoles());
        return "redirect:/user/list";
    }
```

```
    @GetMapping("/delete/{username}")
    public String deleteTicket(@PathVariable("username") String username) {
        umService.delete(username);
        return "redirect:/user/list";
    }
}
```

3. Create the following JSP view **/WEB-INF/jsp/listUser.jsp**:

```
<!DOCTYPE html>
<html>
<head><title>Customer Support User Management</title></head>
<body>
<c:url var="logoutUrl" value="/logout"/>
<form action="${logoutUrl}" method="post">
    <input type="submit" value="Log out" />
    <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
</form>

<br /><br />

<a href="<c:url value="/ticket" />">Return to list tickets</a>

<h2>Users</h2>

<a href="<c:url value="/user/create" />">Create a User</a><br /><br />

<c:choose>
    <c:when test="${fn:length(ticketUsers) == 0}">
        <i>There are no users in the system.</i>
    </c:when>
    <c:otherwise>
        <table>
            <tr>
                <th>Username</th><th>Password</th><th>Roles</th><th>Action</th>
            </tr>
            <c:forEach items="${ticketUsers}" var="user">
                <tr>
                    <td>${user.username}</td>
                    <td>${fn:substringAfter(user.password, '{noop}')}</td>
                    <td>
                        <c:forEach items="${user.roles}" var="role" varStatus="status">
                            <c:if test="${!status.first}">, </c:if>
                            ${role.role}
                        </c:forEach>
                    </td>
                    <td>
                        [<a href="<c:url value="/user/delete/${user.username}" />">Delete</a>]
                    </td>
                </tr>
            </c:forEach>
        </table>
    </c:otherwise>
</c:choose>
</body>
</html>
```

Note that when displaying a user's password **user.password**, we remove **{noop}** by using **${fn:substringAfter(user.password, '{noop}')}**.

4. Create the JSP view **/WEB-INF/jsp/addUser.jsp**:

```
<!DOCTYPE html>
<html>
<head><title>Customer Support</title></head>
<body>
<c:url var="logoutUrl" value="/logout"/>
<form action="${logoutUrl}" method="post">
    <input type="submit" value="Log out"/>
    <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
</form>

<h2>Create a User</h2>
<form:form method="POST" modelAttribute="ticketUser">
    <form:label path="username">Username</form:label><br/>
    <form:input type="text" path="username"/><br/><br/>
    <form:label path="password">Password</form:label><br/>
    <form:input type="text" path="password"/><br/><br/>
    <form:label path="roles">Roles</form:label><br/>
    <form:checkbox path="roles" value="ROLE_USER"/>ROLE_USER
    <form:checkbox path="roles" value="ROLE_ADMIN"/>ROLE_ADMIN
    <br/><br/>
    <input type="submit" value="Add User"/>
</form:form>
</body>
</html>
```

5. We will use ***Spring Security***'s JSP taglib. Add the following JSP taglib directive to **base.jspf**.

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

6. In the JSP view **list.jsp**, add a link for admin to access the user management (highlighted in red):

```
<h2>Tickets</h2>
<security:authorize access="hasRole('ADMIN')">
  <a href="<c:url value="/user" />">Manage User Accounts</a><br /><br />
</security:authorize>
<a href="<c:url value="/ticket/create" />">Create a Ticket</a><br/><br/>
```

7. Build and deploy the web application. An admin can add or remove user accounts now. Follow Lecture 8 Slides 28-29 to view the database table contents.

Task 3: Adding custom login, logout, remember-me and Securing web pages and JSP contents

Note that users can still access the user management page by typing the URL pattern directly. We will secure web pages and JSP contents. We will also customize login page, logout, and remember-me.

1. First, we will add the editing ticket functionality to the web app. Add the following `updateTicket` method to the **TicketService** in the **dao** package `hkmu.comps380f.dao` :

```
@Transactional(rollbackFor = TicketNotFound.class)
public void updateTicket(long id, String subject,
                         String body, List<MultipartFile> attachments)
        throws IOException, TicketNotFound {
    Ticket updatedTicket = tRepo.findById(id).orElse(null);
    if (updatedTicket == null) {
        throw new TicketNotFound(id);
    }
    updatedTicket.setSubject(subject);
    updatedTicket.setBody(body);
```

```
    for (MultipartFile filePart : attachments) {
        Attachment attachment = new Attachment();
        attachment.setName(filePart.getOriginalFilename());
        attachment.setMimeContentType(filePart.getContentType());
        attachment.setContents(filePart.getBytes());
        attachment.setTicket(updatedTicket);
        if (attachment.getName() != null && attachment.getName().length() > 0
                && attachment.getContents() != null
                && attachment.getContents().length > 0) {
            updatedTicket.getAttachments().add(attachment);
        }
    }
    tRepo.save(updatedTicket);
}
```

2. We want to secure the customer support application, as follows:
   - ROLE_ADMIN: Can edit and delete any ticket.
   - ROLE_USER: Can only edit ticket created by themselves, and cannot delete any ticket.

   Add the following two controller methods to **TicketController** for editing a ticket. Note how we enforce the above security policy with the codes highlighted in red color.

```
@GetMapping("/edit/{ticketId}")
public ModelAndView showEdit(@PathVariable("ticketId") long ticketId,
                             Principal principal, HttpServletRequest request)
        throws TicketNotFound {
    Ticket ticket = tService.getTicket(ticketId);
    if (ticket == null
            || (!request.isUserInRole("ROLE_ADMIN")
            && !principal.getName().equals(ticket.getCustomerName()))) {
        return new ModelAndView(new RedirectView("/ticket/list", true));
    }

    ModelAndView modelAndView = new ModelAndView("edit");
    modelAndView.addObject("ticket", ticket);

    Form ticketForm = new Form();
    ticketForm.setSubject(ticket.getSubject());
    ticketForm.setBody(ticket.getBody());
    modelAndView.addObject("ticketForm", ticketForm);

    return modelAndView;
}

@PostMapping("/edit/{ticketId}")
public String edit(@PathVariable("ticketId") long ticketId, Form form,
                   Principal principal, HttpServletRequest request)
        throws IOException, TicketNotFound {
    Ticket ticket = tService.getTicket(ticketId);
    if (ticket == null
            || (!request.isUserInRole("ROLE_ADMIN")
            && !principal.getName().equals(ticket.getCustomerName()))) {
        return "redirect:/ticket/list";
    }

    tService.updateTicket(ticketId, form.getSubject(),
            form.getBody(), form.getAttachments());
    return "redirect:/ticket/view/" + ticketId;
}
```

There are more formal ways to secure methods in Spring Security, which is out of the scope of this course. You may check the following documentation for more details:
https://docs.spring.io/spring-security/reference/servlet/authorization/method-security.html

3. Add the JSP view page **edit.jsp** for editing a ticket and adding more attachments to the ticket.

```
<!DOCTYPE html>
<html>
<head>
    <title>Customer Support</title>
</head>
<body>
<c:url var="logoutUrl" value="/logout"/>
<form action="${logoutUrl}" method="post">
    <input type="submit" value="Log out" />
    <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
</form>

<h2>Edit Ticket #${ticket.id}</h2>
<form:form method="POST" enctype="multipart/form-data"
modelAttribute="ticketForm">
    <form:label path="subject">Subject</form:label><br/>
    <form:input type="text" path="subject" /><br/><br/>
    <form:label path="body">Body</form:label><br/>
    <form:textarea path="body" rows="5" cols="30" /><br/><br/>
    <b>Add more attachments</b><br />
    <input type="file" name="attachments" multiple="multiple"/><br/><br/>
    <input type="submit" value="Save"/><br/><br/>
</form:form>
<a href="<c:url value="/ticket" />">Return to list tickets</a>
</body>
</html>
```

4. Add the custom login JSP page **login.jsp**:

```
<!DOCTYPE html>
<html>
<head>
    <title>Customer Support Login</title>
</head>
<body>
<c:if test="${param.error != null}">
    <p>Login failed.</p>
</c:if>
<c:if test="${param.logout != null}">
    <p>You have logged out.</p>
</c:if>
<h2>Customer Support Login</h2>
<form action="login" method="POST">
    <label for="username">Username:</label><br/>
    <input type="text" id="username" name="username"/><br/><br/>
    <label for="password">Password:</label><br/>
    <input type="password" id="password" name="password"/><br/><br/>
    <input type="checkbox" id="remember-me" name="remember-me"/>
    <label for="remember-me">Remember me</label><br/><br/>
    <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
    <input type="submit" value="Log In"/>
</form>
</body>
</html>
```

5. In **IndexController**, add the following controller method:

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

6. Modify the JSP page **list.jsp** to show the edit and delete links to suitable users (code in red).

```
(customer: <c:out value="${entry.customerName}"/>)
<security:authorize access="hasRole('ADMIN') or
                            principal.username=='${entry.customerName}'">
  [<a href="<c:url value="/ticket/edit/${entry.id}" />">Edit</a>]
</security:authorize>
<security:authorize access="hasRole('ADMIN')">
  [<a href="<c:url value="/ticket/delete/${entry.id}" />">Delete</a>]
</security:authorize>
<br />
```

7. **Your task:** Modify the JSP page view.jsp similarly as in Step 6.

8. Create the following configuration class in a new package package `hkmu.comps380f.config` :

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http)
            throws Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/user/**").hasRole("ADMIN")
                .requestMatchers("/ticket/delete/**").hasRole("ADMIN")
                .requestMatchers("/ticket/**").hasAnyRole("USER", "ADMIN")
                .anyRequest().permitAll()
            )
            .formLogin(form -> form
                .loginPage("/login")
                .failureUrl("/login?error")
                .permitAll()
            )
            .logout(logout -> logout
                .logoutUrl("/logout")
                .logoutSuccessUrl("/login?logout")
                .invalidateHttpSession(true)
                .deleteCookies("JSESSIONID")
            )
            .rememberMe(remember -> remember
                .key("uniqueAndSecret")
                .tokenValiditySeconds(86400)
            )
            .httpBasic(withDefaults());
        return http.build();
    }
}
```

Note that the request matching in `.authorizeHttpRequests()` are done in the declaration order. If **`.anyRequest().permitAll()`** is put first, other **`.requestMatchers(…).hasRole(…)`** do not work.

9. Build and deploy the web app. Has our security policy been properly enforced?
Think about the URL pattern: **/{ticketId}/delete/{attachment:.+}**