GRAND CANYON UNIVERSITY

CST-339 Activity 6 Guide

Contents

Part 1: Securing a Web Application using an In-Memory Data Store

Overview

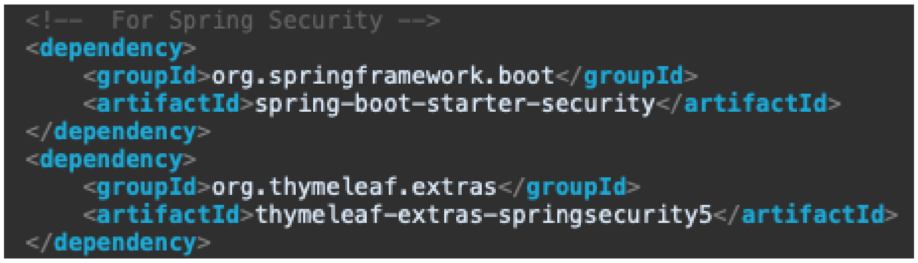
Goal and Directions:

In this activity, you will use Spring Security to support securing your application. An in-memory data store will be used for the credential store. In the next part of the activity, you will change this to use a database.

Execution

Execute this assignment according to the following guidelines:

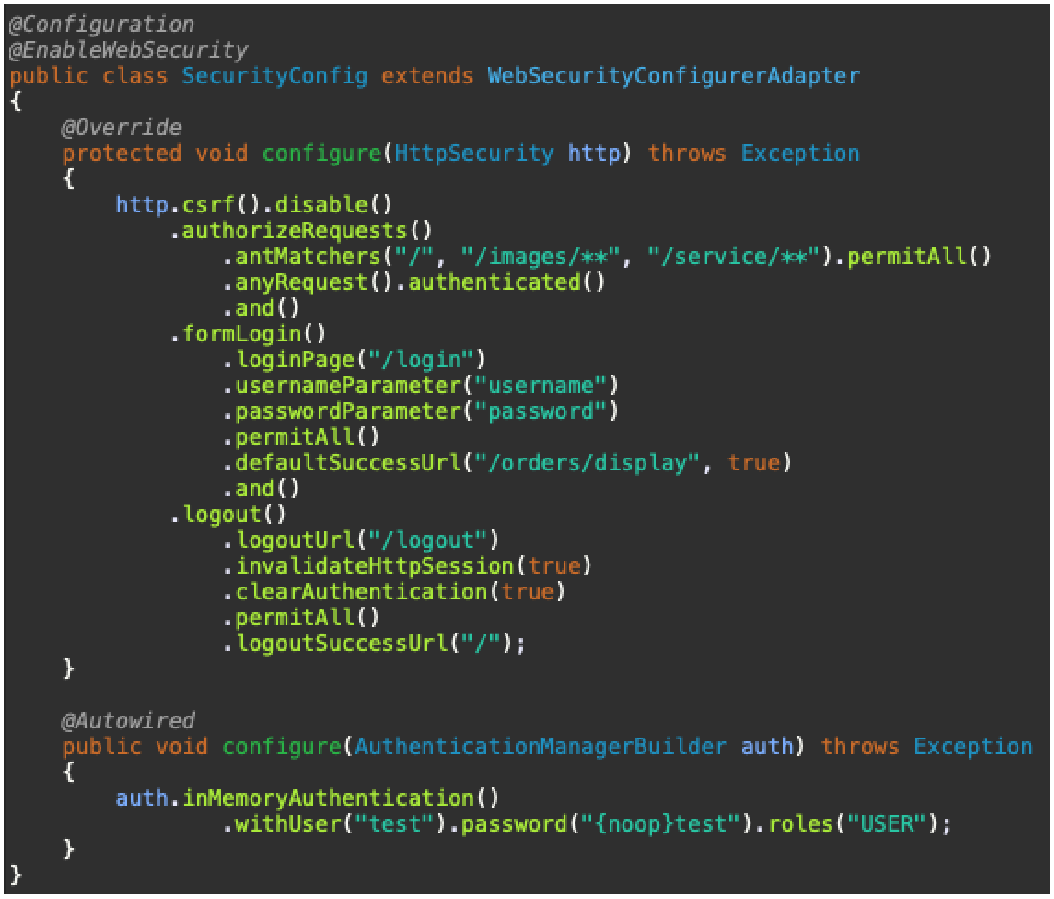
1. Make a copy of the *topic5-1* project by right-clicking on the project, selecting the Copy menu option, then right-clicking in the Workspace, and then selecting the Paste menu option. Name the project *topic6-1*. Rename the Spring application file from *Topic51Application.java* to *Topic61Application.java*. Fixup the *artifactId* and *name* tags in the POM file (pom.xml) so the value is set to *topic6-1*. Run the project to ensure that everything is working properly.
2. Make the following changes to the project:
    a. Add the *spring-boot-starter-security* and *thymeleaf-extras-springsecurity5* dependencies to the POM file.



```
<!--  For Spring Security -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-springsecurity5</artifactId>
</dependency>
```

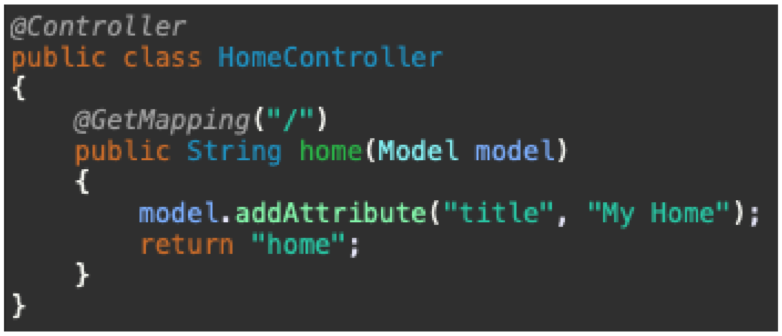
    b. Create a new class named *SecurityConfig* in the *com.gcu* package. This class should extend from the *WebSecurityConfigurerAdapter* class from Spring Security framework class in the *org.springframework.security* package. Mark the class with the *@Configuration* and the @EnableWebSecurity annotations. Override the configure() method and autowire an instance of

AuthenticationManagerBuilder as shown below. This will enable Web Security in the application, set up the paths of root path, static resources, and REST API's to not be protected resources, setup the login form, setup the logout logic, and to use an in-memory datastore for a user with a credentials of test and test.



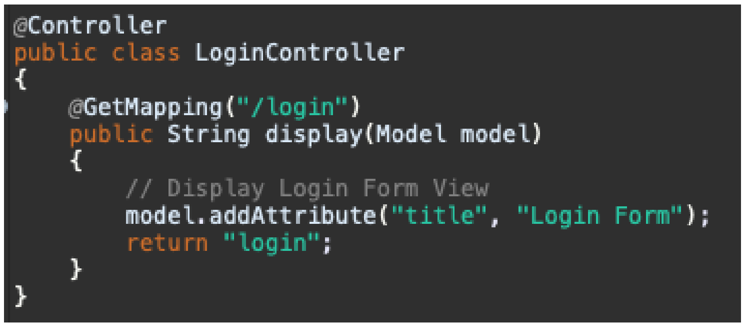
```java
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter
{
    @Override
    protected void configure(HttpSecurity http) throws Exception
    {
        http.csrf().disable()
            .authorizeRequests()
                .antMatchers("/", "/images/**", "/service/**").permitAll()
                .anyRequest().authenticated()
                .and()
            .formLogin()
                .loginPage("/login")
                .usernameParameter("username")
                .passwordParameter("password")
                .permitAll()
                .defaultSuccessUrl("/orders/display", true)
                .and()
            .logout()
                .logoutUrl("/logout")
                .invalidateHttpSession(true)
                .clearAuthentication(true)
                .permitAll()
                .logoutSuccessUrl("/");
    }

    @Autowired
    public void configure(AuthenticationManagerBuilder auth) throws Exception
    {
        auth.inMemoryAuthentication()
                .withUser("test").password("{noop}test").roles("USER");
    }
}
```

c. Create a new *HomeController* class with a route to the root of */* that returns a Home View at *home*. Create a new *home.html* page that uses a layout with a title model attribute of *title* set to *My Home*, displays simple *Welcome Message*, and has a hyperlink set to the URI of */login* (this will be routed to the Login Controller coded next and will be called by the Spring Security framework if a user is not authenticated).



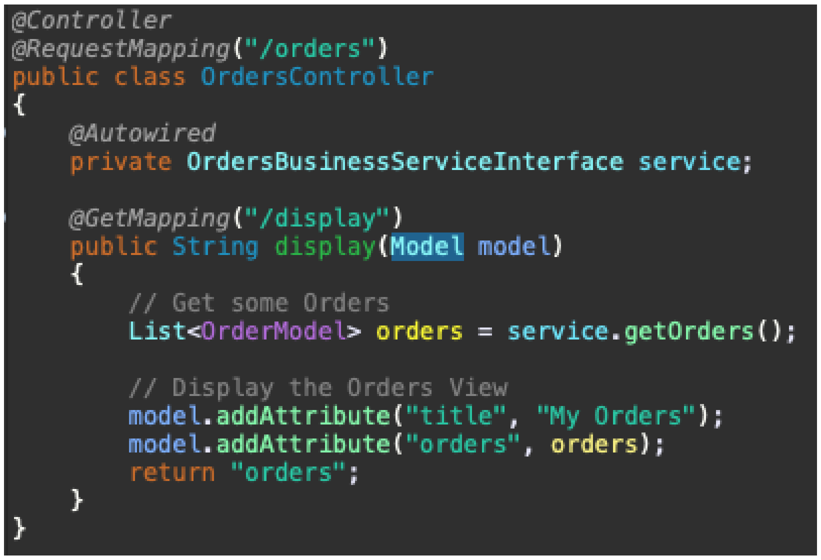
```java
@Controller
public class HomeController
{
    @GetMapping("/")
    public String home(Model model)
    {
        model.addAttribute("title", "My Home");
        return "home";
    }
}
```

d. Update the *LoginController* class by removing class request mapping to */login*. Remove the *doLogin()* method (but save this code for the next step). Update the *display()* method with a new route to */login* route and return the Login View named *login*. Delete the *LoginModel* from the project. This Controller will be called by the Spring Security Config class when a user needs to be authenticated.



```java
@Controller
public class LoginController
{
    @GetMapping("/login")
    public String display(Model model)
    {
        // Display Login Form View
        model.addAttribute("title", "Login Form");
        return "login";
    }
}
```

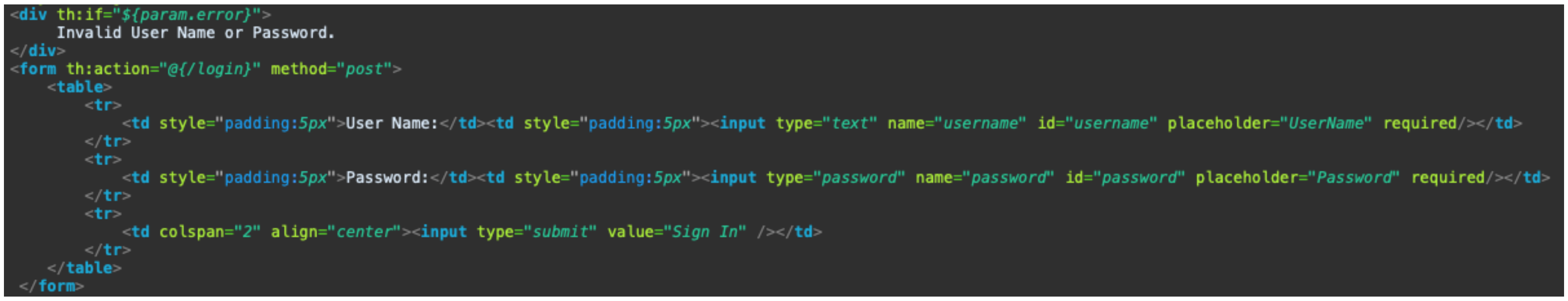
e. Create a new *OrdersController* class with route to */orders*. Copy the code from *doLogin()* from the Login Controller to a *displayOrders(Model model)* method with a route set to */display*. Declare an instance of an *OrdersBusinessService* named *service* as a class member variable and autowire this into the class. Remove any Data Validation logic and test code so that just the *OrdersBusinessService.getOrders()* method is called with navigation passed to the Orders View named *orders*.



```java
@Controller
@RequestMapping("/orders")
public class OrdersController
{
    @Autowired
    private OrdersBusinessServiceInterface service;

    @GetMapping("/display")
    public String display(Model model)
    {
        // Get some Orders
        List<OrderModel> orders = service.getOrders();

        // Display the Orders View
        model.addAttribute("title", "My Orders");
        model.addAttribute("orders", orders);
        return "orders";
    }
}
```

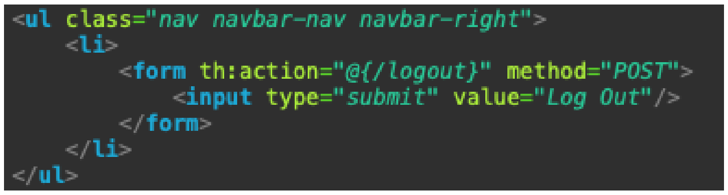
f. Remove the *LoginModel* class from the project.

g. Update the *login.html* page to *post* to the *@{/login}* Thymeleaf action (this is per the Spring Security framework requirements). Remove all the field and data validation code. Add *required* attributes to the input username and password text controls. Add a *div* tag that displays a message *Invalid User Name or Password* using a Thymeleaf condition of *${param.error}*.



```
<div th:if="${param.error}">
     Invalid User Name or Password.
</div>
<form th:action="@{/login}" method="post">
    <table>
        <tr>
            <td style="padding:5px">User Name:</td><td style="padding:5px"><input type="text" name="username" id="username" placeholder="UserName" required/></td>
        </tr>
        <tr>
            <td style="padding:5px">Password:</td><td style="padding:5px"><input type="password" name="password" id="password" placeholder="Password" required/></td>
        </tr>
        <tr>
            <td colspan="2" align="center"><input type="submit" value="Sign In" /></td>
        </tr>
    </table>
 </form>
```

h. Update the NavBar in the *common.html* page fragment to replace the Login Link with a Log Out button that posts to the */logout* Thymeleaf action (this will invoke the log out logic configured in the Spring Security Config file).



```
<ul class="nav navbar-nav navbar-right">
    <li>
        <form th:action="@{/logout}" method="POST">
            <input type="submit" value="Log Out"/>
        </form>
    </li>
</ul>
```

3. Run the application. Open a browser and go to *localhost:8080/*. Click the hyperlink to display the login form. Submit the login form using a username of *test* and a password of *test* to display the Orders page. Verify that the Orders are displayed correctly from the database. Take a screenshot of the Orders page. Also, test the REST API /getjson and /getxml API's, which currently are anonymous. Take a screenshot of the JSON and XML responses.
4. Update the POM file to name the output JAR file *cst339activity*. Run a Maven build and test the JAR file out from a terminal window. Reference the steps as needed from the Activity 1 Guide.
5. For extra practice (not graded), complete the implementation of the *findById()*, create(), *update()* and *delete()* methods of the *UsersDataService* and test them within your application.

<u>Deliverables</u>:

The following needs to be submitted as this part of the activity:

a. Screenshot of the Orders page.
b. Screenshots of the JSON and XML REST API responses.

Part 2: Securing a Web Application using a Database

Overview

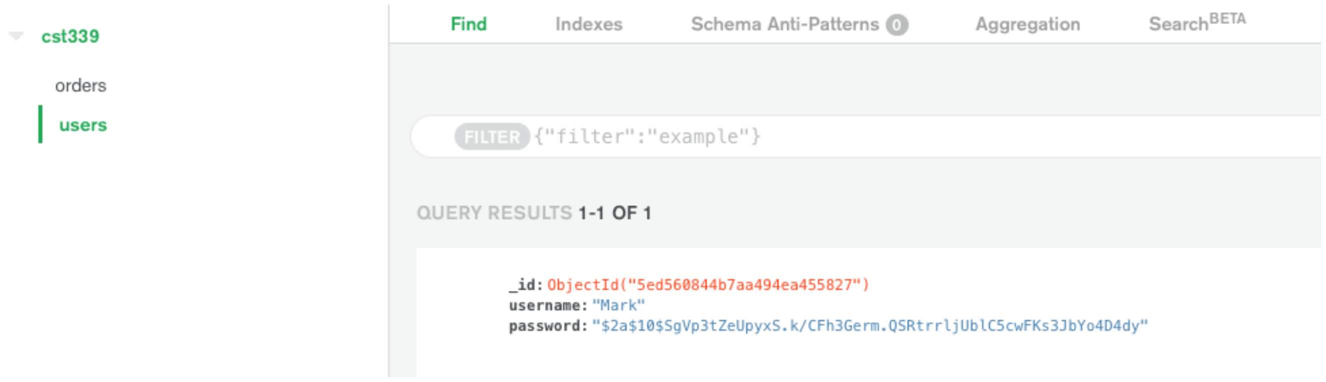
Goal and Directions:

In this activity, you will use Spring Security to support securing your application. The MongoDB Atlas database will be used for the credential store.

Execution

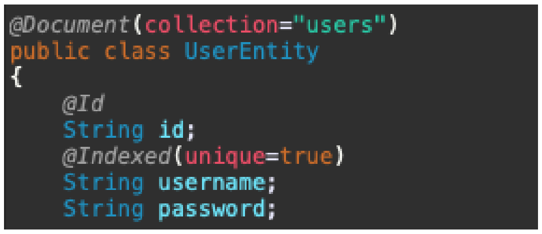
Execute this assignment according to the following guidelines:

1. Make a copy of the *topic6-1* project by right-clicking on the project, selecting the Copy menu option, then right-clicking in the Workspace, and then selecting the Paste menu option. Name the project *topic6-2*. Rename the Spring application file from *Topic61Application.java* to *Topic62Application.java*. Fix up the *artifactId* and *name* tags in the POM file (pom.xml) so the value is set to *topic6-2*. Run the project to ensure that everything is working properly.
2. Make the following changes to the project:
    a. Log into your MongoDB Atlas account. Add a new User Collection in MongoDB Atlas with an id, username, and password properties all of type String. Populate a test user.



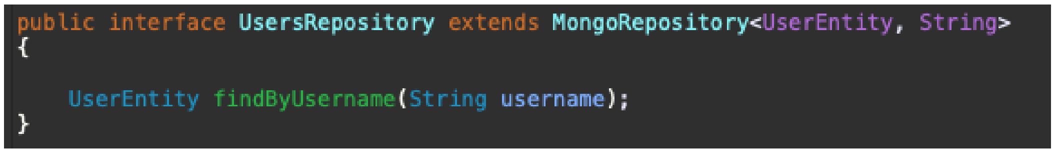
```
_id: ObjectId("5ed560844b7aa494ea455827")
username: "Mark"
password: "$2a$10$SgVp3tZeUpyxS.k/CFh3Germ.QSRtrrljUblC5cwFKs3JbYo4D4dy"
```

    b. Create a new *UserEntity* class in the *com.gcu.data.entity* package with an *id, username, and password* properties of type String with all getters and setters. Create default and non-default constructors. Mark the *id* with the *@Id* annotation. Mark the *username* with the *@Indexed(unique=true)* annotation. Mark the *@Document* annotation at the class level and set the collection attribute value to *users*.



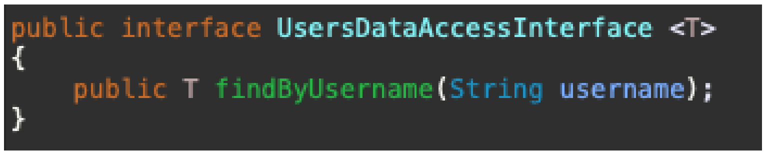
```
@Document(collection="users")
public class UserEntity
{
    @Id
    String id;
    @Indexed(unique=true)
    String username;
    String password;
```

c. Create a new *UsersRepository* class *com.gcu.data.repository* package that extends from the *MongoRepository* and associated with the *UserEntity* class. Add a new method declaration *UserEntity findByUsername(String username)* (this will be used to find a user in the database during authentication).



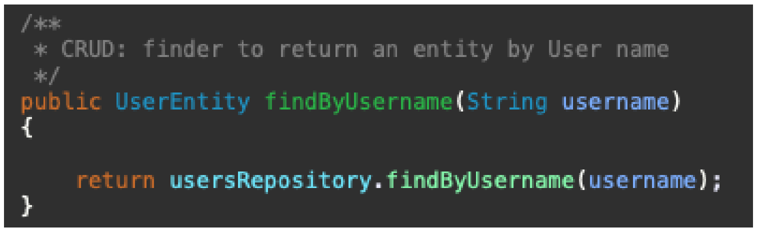
```
public interface UsersRepository extends MongoRepository<UserEntity, String>
{

    UserEntity findByUsername(String username);
}
```

d. Create a new *UsersDataAccessInterface* interface class using a Generic at the class level set to *T* with a method declaration of *T findByUsername(String username)*.



```
public interface UsersDataAccessInterface <T>
{
    public T findByUsername(String username);
}
```

e. Create a new *UsersDataService* class that implements both the *UsersDataAccessInterface* and *DataAccessInterface* interfaces, specifying a Generic type of *UserEntity*. Declare a class scope method variable named *usersRepository* of type *UsersRepository* that is autowired using constructor injection. For now, only implement the *findByUsername()* method and in the implementation call the *usersRepository.findByUsername()* method. Mark with the *@Service* annotation at the class level.



```
/**
 * CRUD: finder to return an entity by User name
 */
public UserEntity findByUsername(String username)
{

    return usersRepository.findByUsername(username);
}
```

f. Add a new *UserBusinessService* class *com.gcu.business* package that implements the implements the *UserDetailsService* interface (from within the *org.springframework.security* package), which is required by the Spring Security framework to support authentication. Mark the class with the @Service annotation. Declare a class scope method variable named *service* of type *UsersDataService* that is autowired using constructor injection. Override and implement *loadUserByUsername()* method, which will be called by the Spring Security framework to support authentication, using the *UsersDataService* as follows.

```
/**
 * This method is overridden from the base class and is used to support Spring Security user authentication
 */
@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException
{
    // Try to find the User in the database. If not found throw a User Not Found exception else return a Spring Security User.
    UserEntity user = service.findByUsername(username);
    if(user != null)
    {
        List<GrantedAuthority> authorities = new ArrayList<GrantedAuthority>();
        authorities.add(new SimpleGrantedAuthority("USER"));
        return new User(user.getUsername(), user.getPassword(), authorities);
    }
    else
    {
        throw new UsernameNotFoundException("username not found");
    }
}
```

g. Delete the *SecurityBusinessService* class from the project.

h. Update the *SecurityConfig* class configure() method by removing the code for using memory authentication. Declare a class scope method variable named *service* of type *UserBusinessService* that is autowired. Declare a class scope method variable named *passwordEncoder* of type *PasswordEncoder* from within the *org.springframework.security.crypto* package that is autowired. Add a SpringBean definition in a method named passwordEncoder() that returns instance of *BCryptPasswordEncoder*. Update the autowired configuration() method to set the *userDetailsServic*() and the *passwordEncoder*() on the *auth* parameter as follows:

```
@Autowired
PasswordEncoder passwordEncoder;

@Autowired
UserBusinessService service;

@Bean
BCryptPasswordEncoder passwordEncoder()
{
    return new BCryptPasswordEncoder();
}
```

```
@Autowired
public void configure(AuthenticationManagerBuilder auth) throws Exception
{
    auth
    .userDetailsService(service)
    .passwordEncoder(passwordEncoder);
}
```

i. Encrypt the password created in Step 2a using the following java code and print the encrypted password to the console. Update the Users Collection accordingly to set the password to the encrypted value.

String encoded = new BCryptPasswordEncoder().encode(plainTextPassword);

3. Run the application. Open a browser and go to *localhost:8080/*. Click the hyperlink to display the login form. Submit the login form using a username of and password entered

in the MongoDB Atlas Users Collection to display the Orders page. Verify that the Orders are displayed correctly from the database. Take a screenshot of the Orders page. Also, test the REST API /getjson and /getxml API's, which are currently anonymous. Take a screenshot of the JSON and XML responses.

4. Update the POM file to name the output JAR file *cst339activity*. Run a Maven build and test the JAR file out from a terminal window. Reference the steps as needed from the Activity 1 Guide.
5. For extra practice (not graded), complete the implementation of the *findById(),* create(), *update()* and *delete()* methods of the *UsersDataService* and test them within your application.

Deliverables:

The following needs to be submitted as this part of the activity:

a. Screenshot of the Orders page.
b. Screenshots of the JSON and XML REST API responses.

Part 3: Securing REST API’s using Basic HTTP Authentication

Overview

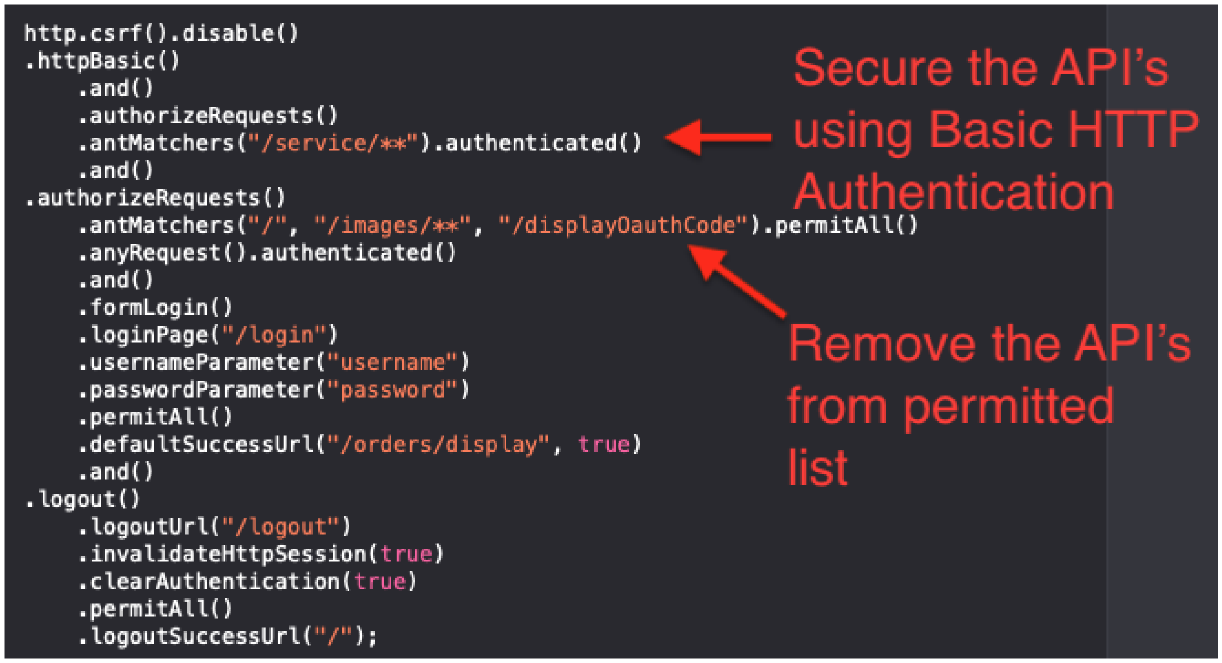
Goal and Directions:

In this activity, you will use Spring Security to support securing your REST API’s. Basic HTTP Authentication will be used in the implementation.

Execution

Execute this assignment according to the following guidelines:

1. Make a copy of the *topic6-2* project by right-clicking on the project, selecting the Copy menu option, then right-clicking in the Workspace, and then selecting the Paste menu option. Name the project *topic6-3*. Rename the Spring application file from *Topic62Application.java* to *Topic63Application.java*. Fix up the *artifactId* and *name* tags in the POM file (pom.xml) so the value is set to *topic6-3*. Run the project to ensure that everything is working properly.
2. Make the following changes to the project by updating the *SecurityConfig* class:
   a. Remove the /service/** from the list of projected URI’s.
   b. Add basic http authentication to the REST API’s at the */service* endpoints.



```
http.csrf().disable()
.httpBasic()
    .and()
    .authorizeRequests()
    .antMatchers("/service/**").authenticated()
    .and()
.authorizeRequests()
    .antMatchers("/", "/images/**", "/displayOauthCode").permitAll()
    .anyRequest().authenticated()
    .and()
    .formLogin()
    .loginPage("/login")
    .usernameParameter("username")
    .passwordParameter("password")
    .permitAll()
    .defaultSuccessUrl("/orders/display", true)
    .and()
.logout()
    .logoutUrl("/logout")
    .invalidateHttpSession(true)
    .clearAuthentication(true)
    .permitAll()
    .logoutSuccessUrl("/");
```

Secure the API's using Basic HTTP Authentication

Remove the API's from permitted list

3. Run the application. Open a browser and go to *localhost:8080/* and test the REST API's at */getjson* and */getxml* API's, which will now require authentication. The user will be taken to the application's login screen.
4. Test the REST API's at */getjson* and */getxml* API's using Postman. This can be setup as follows:
    a. Create a new Collection in Postman named *CST-339* by selecting the *Collections* tab and clicking the + *New Collection* button.
    b. Create a new Request in the Collection by selecting *Add Request* from the Collection context menu. Name your request *Get Products as JSON* and click the *Save* button.
    c. Select a GET request and enter the URL for the */getjson* API.
    d. Select the *Authorization* tab. From the *Type* dropdown select the *Basic Auth* type. Enter a valid *username* and *password* that can be used to also login in the web application.
    e. Click the *Save* button.
    f. Click the *Send* button. Take a screen shot of the formatted JSON output.
    g. Select the *Authorization* tab. Enter an invalid username or password. Click the *Send* button. Take a screenshot of the 401 Unauthorized code in the response.
    h. Repeat steps b through g for the */getxml* API.

<u>Deliverables</u>:

The following needs to be submitted as this part of the activity:

    a. Screenshot of the */getjson* API response with good login credentials.
    b. Screenshot of the */getjson* API response with bad login credentials.
    c. Screenshot of the */getxml* API response with good login credentials.
    d. Screenshot of the */getxml* API response with bad login credentials.

Part 4: Securing REST API's using OAuth2 Authentication

Overview

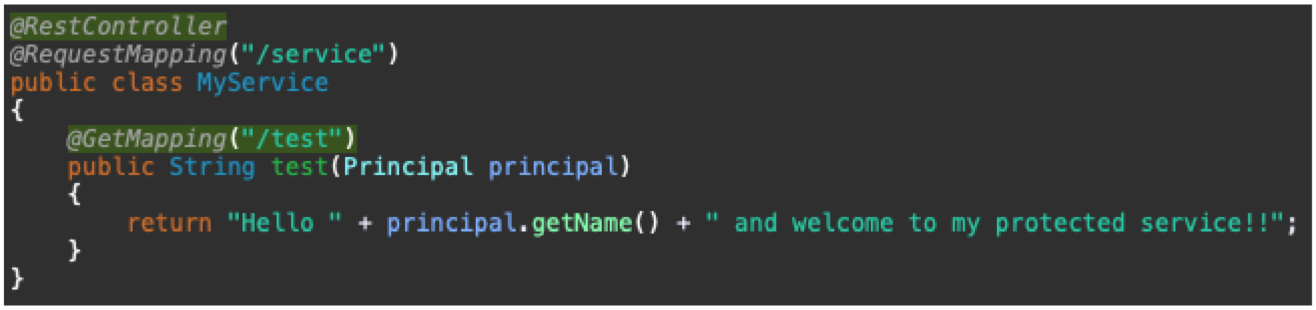
Goal and Directions:

In this activity, you will use Spring Security to support securing a REST API. OAuth2 Authentication with GitHub will be used in the implementation.

Execution

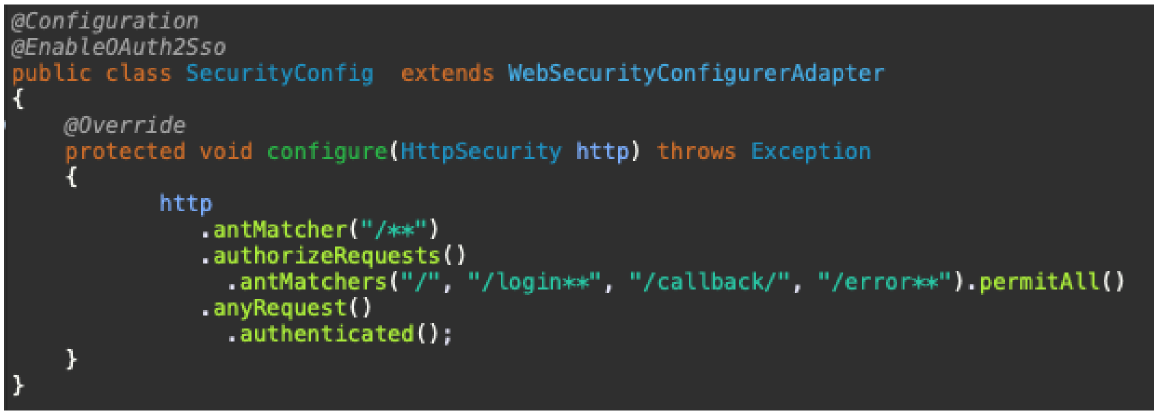
Execute this assignment according to the following guidelines:

1. Setup GitHub as an OAuth2 provider as follows:
    a. Log into your GitHub account.
    b. Go to your GitHub Profile and click the *Settings* menu option.
    c. Click the *Developer Settings* link.
    d. Click the *OAuth Apps* link.
    e. Click the *New OAuth App* button.
    f. Specify the following settings:
        i. Application Name: CST-339 Activity
        ii. Homepage URL: http://localhost:8080/
        iii. Authorization callback URL: http://localhost:8080/
    g. Click the *Register Application* button.
    h. Note the *Client ID* and the *Client Secret* that you will need in your application configuration.
2. Create a new Spring Boot Starter project named *topic6-4*. Add the Spring Web, Spring Boot DevTools, Cloud Security, and Cloud OAuth2 dependencies to the project.
3. Add a new REST service class named *MyService* to the project. The class should be marked with the *@RestController* annotation and *@RequestMapping("/service")* annotation. Add an API named *test()* that is marked with the *@GetMapping("/test")* annotation that takes a *Principal* as a method argument. The implementation should print a "hello" method that also prints the authenticated user's name (by calling the *getName()* on the *Principal* method argument).



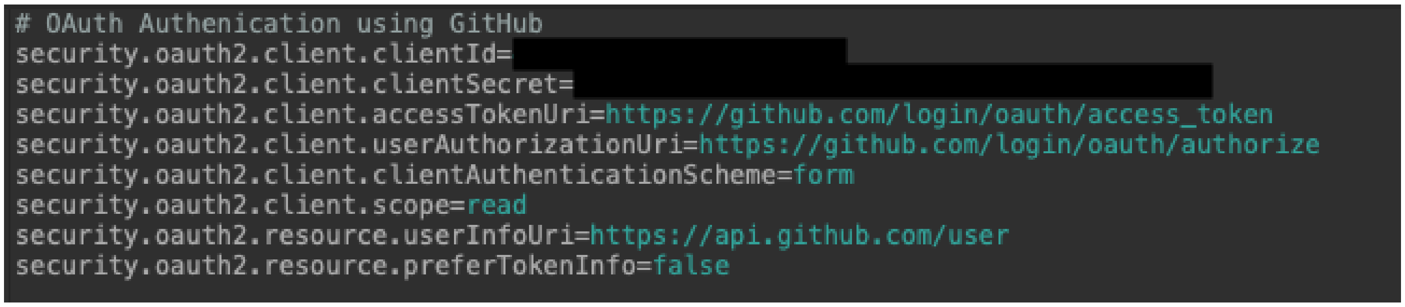
```
@RestController
@RequestMapping("/service")
public class MyService
{
    @GetMapping("/test")
    public String test(Principal principal)
    {
        return "Hello " + principal.getName() + " and welcome to my protected service!!";
    }
}
```

4. Create a new class named *SecurityConfig*, which will be used to configure the OAuth2 access rules. The class should be marked with the *@Configuration* annotation and *@EnableOAuth2Sso* annotation. Override the *configure()* method and set the configuration so the /, /login, /callback, and /error URI’s are all permitted and all other URI’s needing authentication.



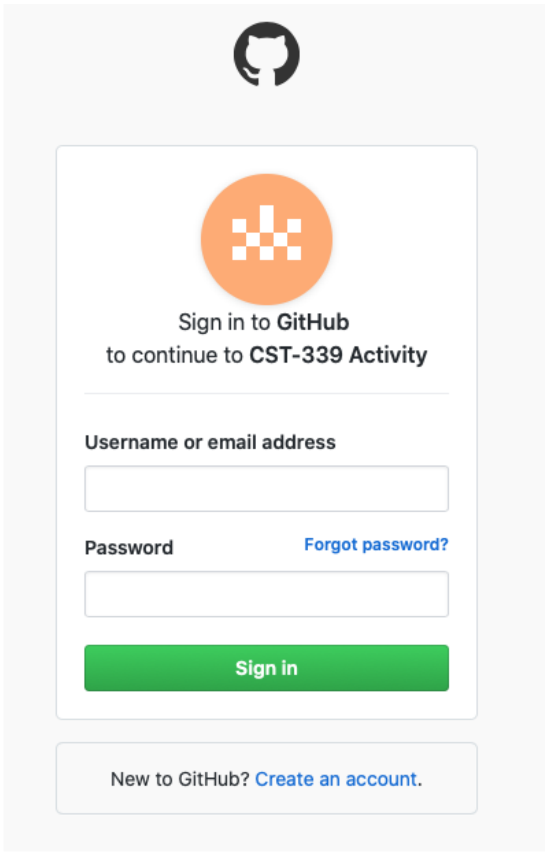
```
@Configuration
@EnableOAuth2Sso
public class SecurityConfig  extends WebSecurityConfigurerAdapter
{
    @Override
    protected void configure(HttpSecurity http) throws Exception
    {
            http
               .antMatcher("/**")
               .authorizeRequests()
                 .antMatchers("/", "/login**", "/callback/", "/error**").permitAll()
               .anyRequest()
                 .authenticated();
    }
}
```

5. Add the following security configuration properties to the *application.properties* file.



```
# OAuth Authenication using GitHub
security.oauth2.client.clientId=[redacted]
security.oauth2.client.clientSecret=[redacted]
security.oauth2.client.accessTokenUri=https://github.com/login/oauth/access_token
security.oauth2.client.userAuthorizationUri=https://github.com/login/oauth/authorize
security.oauth2.client.clientAuthenticationScheme=form
security.oauth2.client.scope=read
security.oauth2.resource.userInfoUri=https://api.github.com/user
security.oauth2.resource.preferTokenInfo=false
```

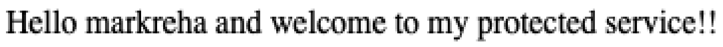
6. Run the application. Open a browser and go to *localhost:8080/service/test* and test the REST API, which will now require OAuth2 authentication using your GitHub credentials. The first time you access the API, you will be presented with login screen where you will need to enter your GitHub credentials. Take a screenshot of the GitHub login screen. Take another screenshot of the API response after you have logged using your GitHub credentials. To reauthenticate in your browser, you will need to clear all the cookies from github.com.


Sign in to GitHub
to continue to CST-339 Activity

Username or email address

Password

New to GitHub? Create an account.


Hello markreha and welcome to my protected service!!

Deliverables:

The following needs to be submitted as this part of the activity:

a. Screenshot of the GitHub login screen.
b. Screenshot of the API response once authenticated using GitHub.

Research Questions

1. Research Questions: For traditional ground students, answer the following questions in a Microsoft Word document:
   a. Research the forms based authentication scheme. Describe how this works. Why it is important to use the Spring Security framework versus developing your own custom security framework?
   b. Research the Basic HTTP authentication schema. Describe how this works. How does this technology help secure a REST API endpoint?

Final Activity Submission

1. In a Microsoft Word document, complete the following for the Activity Report:

a. Cover Sheet with the name of this assignment, date, and your name.
b. Section with a title that contains all the screenshots for each part of the activity.
c. Section with a title that contains the answers to the research questions (traditional ground students only).

2. Submit the Activity Report as directed by your instructor.