# Spring Security

## Lab Manual

VMware® Education Services
VMware, Inc.
www.vmware.com/education

Spring Security

Lab Manual

Part Number EDU-EN-SS-LAB (14-JAN-2022)

Copyright © 2022 VMware, Inc. All rights reserved. This manual and its accompanying materials are protected by U.S. and international copyright and intellectual property laws. VMware products are covered by one or more patents listed at http://www.vmware.com/go/patents. VMware is a registered trademark or trademark of VMware, Inc. in the United States and/or other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies. VMware Go™, VMware Verify™ are registered trademarks or trademarks of VMware, Inc. in the United States and/or other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies.

The training material is provided "as is," and all express or implied conditions, representations, and warranties, including any implied warranty of merchantability, fitness for a particular purpose or noninfringement, are disclaimed, even if VMware, Inc., has been advised of the possibility of such claims. This training material is designed to support an instructor-led training course and is intended to be used for reference purposes in conjunction with the instructor-led training course.

The training material is not a standalone training tool. Use of the training material for self-study without class attendance is not recommended. These materials and the computer programs to which it relates are the property of, and embody trade secrets and confidential information proprietary to, VMware, Inc., and may not be reproduced, copied, disclosed, transferred, adapted or modified without the express written approval of VMware, Inc.

# Lab 1: Lab Setup

## Objective and Tasks

Set up the lab environment and familiarize yourself with the Reward Dining application:

1. Download the Lab Projects
2. Install the Required Software
3. Import Projects to Your IDE
4. Inspect the Application and Database Schema

The labs in this course teach key concepts in the context of a problem domain. The Reward Dining application provides a real-world context for applying the techniques that you learned to secure business applications.

The domain is called Reward Dining. Customers save money every time they eat at one of the restaurants participating in the network, which is like a Frequent Flyer program for restaurants. For example, Keith wants to save money for his children's education. Every time he dines at a restaurant participating in the network, his account is credited with a reward. The reward will be shared by his two children: Annabelle and Corgan. Thus, Annabelle gets a fund to help her with her college fees.

## Task 1: Download the Lab Projects

The source code for the labs is hosted on GitHub. All lab projects are Maven projects.

1. Go to https://github.com/platform-acceleration-lab/spring-security-code/archive/refs/tags/v1.0.0.zip to download the Zip file containing the labs.
2. Unzip the file to a directory of your choice.

   The unzipped directory, `/lab`, contains the lab projects.

## Task 2: Install the Required Software

You install the required software if you do not have it on your machine.

1. Install JDK 11 or later.
2. Install curl.
3. Install an IDE.

   You can install Spring Tools, IntelliJ IDE, or any other IDE.

## Task 3: Import Projects to Your IDE

The lab directory includes Maven projects. Every lab has two projects: the starter project and the solution. You must use the lab documentation to complete the starter project.

1. Import projects to your IDE.
   a. For Spring Tools, you can import the lab directory, which will import the projects in that directory.

      You can select **Import** from the **File** menu.

   b. For IntelliJ, you can import projects through the parent `pom.xml`, and IntelliJ will automatically set them up as multimodule projects.

      You can select **Import Projects** from the **File** menu.
2. Verify that projects are imported correctly.

   The IDE must not report errors.

## Task 4: Inspect the Application and Database Schema

The `00-rewards-common` project contains all the business logic of the Reward Dining application.

IMPORTANT: All labs in this course depend on this project. You must not change this project.

1. Inspect the schema and the test data in the in-memory database that is used by the application.
   – The database schema is at `src/main/resources/db/schema.sql`.
   – The data populated in the database is at `src/main/resources/db/data.sql`.
2. Inspect the business logic of the Reward Dining application.

   When the application starts, several rewards are generated by the `RewardDiningPopulator` class in the `rewardsdining.rewards` package.

# Lab 2: Introduction to Spring Security

## Objective and Tasks

Use Spring Security in a Spring Boot application and understand the default security configuration:

1. Run the Existing Application
2. Enable Spring Security
3. Log Out of the Application
4. Change the Default User Name and Password
5. Use Basic Authentication
6. Inspect the Default HTTP Headers
7. See the Security Filters in Action
8. Create a Simple Filter

You must use the `20-security-basics` project.

## Task 1: Run the Existing Application

The Reward Network web application is currently not secured. It allows any user to access all available endpoints.

1. Run the application and go to http://localhost:8080.
2. Click the **Accounts** link in the **Admin** menu and ensure that you can access the list of accounts.

   Credentials are not required.

## Task 2: Enable Spring Security

Spring Boot autoconfigures Spring Security when `spring-boot-starter-security` is present on the classpath.

The Spring Security autoconfiguration has been explicitly disabled.

1. Verify the presence of `spring-boot-starter-security` in the parent POM file of the project.

   This file is in the root directory of the projects: `lab/pom.xml`.
2. Remove the exclusion from `SpringIntroApplication`.

   This step autoconfigures Spring Security, and all endpoints are protected by default.
3. Run the application and check the log.

   The autogenerated password appears for the default user.

   NOTE: Spring Boot DevTools has been added to your projects. The application automatically restarts when a change occurs.

```
INFO : o.s.b.a.s.s.UserDetailsServiceAutoConfiguration -

Using generated security password: c954000e-b213-42e8-a63a-f4a05d1a1c88

DEBUG: o.s.s.w.a.e.ExpressionBasedFilterInvocationSecurityMetadataSource - Adding web
access control expression [authenticated] for any request
```

4. Go to http://localhost:8080/accounts.

   The resource is now protected. You are redirected to the login form. The original request is stored in the session. You are redirected to the requested URL after successful authentication.

5. Log in to the Rewards Dining application.
   - User name: user
   - Password: Use the autogenerated password

## Task 3: Log Out of the Application

Spring Security stores `SecurityContext` in the session, which enables you to navigate through the protected resources without providing your credentials on every request. Spring Boot autoconfigures the logout functionality by removing `Authentication` from the current `SecurityContext` and invalidating the HTTP session.

1. Go to http://localhost:8080/logout.

   An autogenerated logout page appears.

2. Click **Logout**.

   Q1. Where are you redirected after logout?

   A1. You are redirected back to the login page with the You have been signed out message.

## Task 4: Change the Default User Name and Password

Every time the application restarts, a new password is generated. Although this method is a good autoconfiguration default, it might not be convenient for your development and testing purposes.

1. Open the `application.properties` user `src/main/resources`.
   a. Add the property `spring.security.user.name` with the value `keith`.
   b. Add the property `spring.security.user.password` with the value `secureP@ssword`.
2. Run the application again and go to http://localhost:8080/accounts with the new credentials.

## Task 5: Use Basic Authentication

Spring Boot autoconfigures form-based and basic authentication. Spring Security uses content negotiation to determine whether to use basic or form-based authentication.

You test basic authentication.

1. Use curl to access http://localhost:8080/accounts without credentials.

   ```
   curl -v http://localhost:8080/accounts
   ```

   Q1. Which HTTP status code appears?

   A1. A 401 Unauthorized code appears. The resource is secured but you have not provided credentials.

   You must be authenticated to authorize the request. The response body is not HTML but JSON.

2. Use curl to access http://localhost:8080/accounts with credentials.

   ```
   curl -v -u keith:secureP@ssword http://localhost:8080/accounts
   ```

   A 200 HTTP status code appears. The accounts are represented in JSON in the response body.

## Task 6: Inspect the Default HTTP Headers

Spring Security adds several headers in the response to protect against common vulnerabilities and adds secure defaults.

1. Inspect the headers from the previous request to http://localhost:8080/accounts.

   The following headers appear among others:

```
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
```

## Task 7: See the Security Filters in Action

Every request passes through a set of security filters that are defined by the Spring Security filter chain. The number of filters in the filter chain depends on your security configuration. `FilterChainProxy` delegates requests to a list of Spring-managed filter beans based on the path.

1. Add a line in `application.properties` to change the log level of the `FilterChainProxy`.

   `logging.level.org.springframework.security.web.FilterChainProxy=trace`

2. Restart the application and go to http://localhost:8080/accounts.
3. Inspect the logs and view the filters involved in the requests.

```
TRACE: o.s.security.web.FilterChainProxy - Invoking WebAsyncManagerIntegrationFilter (1/15)
TRACE: o.s.security.web.FilterChainProxy - Invoking SecurityContextPersistenceFilter (2/15)
DEBUG: o.s.s.w.c.SecurityContextPersistenceFilter - Set SecurityContextHolder to empty
SecurityContext
TRACE: o.s.security.web.FilterChainProxy - Invoking HeaderWriterFilter (3/15)
TRACE: o.s.security.web.FilterChainProxy - Invoking CsrfFilter (4/15)
TRACE: o.s.security.web.FilterChainProxy - Invoking LogoutFilter (5/15)
TRACE: o.s.security.web.FilterChainProxy - Invoking UsernamePasswordAuthenticationFilter
(6/15)
TRACE: o.s.security.web.FilterChainProxy - Invoking DefaultLoginPageGeneratingFilter (7/15)
TRACE: o.s.security.web.FilterChainProxy - Invoking DefaultLogoutPageGeneratingFilter
(8/15)
TRACE: o.s.security.web.FilterChainProxy - Invoking BasicAuthenticationFilter (9/15)
TRACE: o.s.security.web.FilterChainProxy - Invoking RequestCacheAwareFilter (10/15)
TRACE: o.s.security.web.FilterChainProxy - Invoking SecurityContextHolderAwareRequestFilter
(11/15)
TRACE: o.s.security.web.FilterChainProxy - Invoking AnonymousAuthenticationFilter (12/15)
DEBUG: o.s.s.w.a.AnonymousAuthenticationFilter - Set SecurityContextHolder to anonymous
SecurityContext
TRACE: o.s.security.web.FilterChainProxy - Invoking SessionManagementFilter (13/15)
TRACE: o.s.security.web.FilterChainProxy - Invoking ExceptionTranslationFilter (14/15)
TRACE: o.s.security.web.FilterChainProxy - Invoking FilterSecurityInterceptor (15/15)
```

By default, 15 security filters are configured. All endpoints require authentication by default and pass through all filters.

## Task 8: Create a Simple Filter

Spring Security uses Servlet filters to drive authentication, authorization, and so on. You create a simple logging filter.

A `LoggingFilter` class is already created under the `rewardsdining.web` package.

1. Log a debug message before delegating to the next filter in the chain.
2. Delegate to the next filter by calling `doFilter` on the filter chain.
3. Annotate the class with `@Component` to make it a Spring-managed bean.
4. Restart the application and go to http://localhost:8080.

   Your message appears in the log.

   Spring Boot registers Filters configured as Spring-managed beans in the Servlet container automatically. The logging filter is configured with other filters like the Encoding filter (but not part of the Spring Security filter chain).

# Lab 3: Customizing Authentication

## Objective and Tasks

Configure different authentication mechanisms and user storage options:

1. Change the Default Login Page
2. Configure Spring Security to Use the New Login Page
3. Configure Spring Security to Use a Custom Logout Page
4. Define Success URLs Based on the User Role
5. Configure JDBC Authentication
6. Implement UserDetailsService
7. Implement AuthenticationProvider
8. Set Up an Embedded LDAP Server
9. Set Up LdapAuthenticationProvider
10. Use Spring Boot Actuator to See Audit Events

You must use the `30-authentication` project.

## Task 1: Change the Default Login Page

Spring Security generates a default login page if none is provided. A login page that matches the application might be better for consistency.

A template for a custom login page is provided in `src/main/resources/templates/custom-login.html`.

1. Annotate the `AuthenticationController` class under the `rewardsdining.web` package with `@Controller`.
2. Create a `showLogin` method.
   a. Annotate the method with `@GetMapping("/login")`.
   b. Return the `custom-login` String (the new login form).

## Task 2: Configure Spring Security to Use the New Login Page

A controller that can serve the new login page is available. You configure Spring Security to use the controller.

1. Go to the `SecurityConfig` class and change the default values for the login page in `SecurityFilterChain`.
   a. Set the login page to `/login`.
   b. Review the form at `src/main/resources/templates/custom-login.html` and set the user name and password parameters.

      You must verify the text input names for the user name and password.
   c. Set the default success URL to `/restaurants`.

      Users are redirected to this URL after successful authentication.
2. Run the application and verify that http://localhost:8080/login shows the new login page.

   An in-memory authentication with two users has been configured for you.
3. Use keith as the user name and spring as the password.
4. Verify that you are redirected to the success URL.

## Task 3: Configure Spring Security to Use a Custom Logout Page

The default logout page is not configured with the custom login page. A custom logout page is provided in `src/main/resources/templates/custom-logout.html`.

1. Create a `showLogout` method in `AuthenticationController`.
   a. Annotate the method with `@GetMapping("/logout")`.
   b. Return the `custom-logout` String (the new logout form).
2. Go to http://localhost:8080/logout and log out of the application to test the new logout page.

   You are redirected to the login page.

## Task 4: Define Success URLs Based on the User Role

You implement a custom `AuthenticationSuccessHandler` to conditionally decide the URL to redirect the user after authentication.

1. Find the `CustomAuthenticationSuccessHandler` class under the `rewardsdining.security` package.
2. Implement the `determineTargetUrl` method.
   a. Use the `getAuthorities` static method from the `SecurityUtils` class (in the common project) to get the roles of the current user.
   b. Return the success URL.
      - When the user has the admin role, return `/accounts`.
      - When the user has another role, return `/restaurants`.
5. Configure `formLogin` in `SecurityFilterChain` and set `successHandler` to use `CustomAuthenticationSuccessHandler` (create an instance).
6. Run the application and access with keith as the user name and spring as the password.

7. Verify that you are redirected to `/restaurants`.
8. Log out of the application and log in again with chad as the user name and spring as the password.
9. Verify that you are redirected to `/accounts`.

## Task 5: Configure JDBC Authentication

The built-in in-memory authentication might be good during development, but usually you store users in a database. You change the authentication to the built-in JDBC-based authentication.

1. Comment the existing `InMemoryUserDetailsManager` bean definition in the `SecurityConfig` class.
2. Uncomment the `@Bean` annotation from the `jdbcAuthentication` method.
3. Implement the method by creating a new `JdbcUserDetailsManager` and passing the `dataSource` in the constructor.
   a. Set the `usersByUsernameQuery`.

   ```
   select username, password, 'true' as enabled from T_ACCOUNT where username =
   ?
   ```

   b. Set the `authoritiesByUsernameQuery`.

   ```
   select username, role_name as authority from T_ACCOUNT a inner join
   T_ACCOUNT_ROLE ar on (a.id = ar.account_id) where a.username = ?
   ```
6. Run the application and verify that you can log in with keith as the user name and spring as the password.

## Task 6: Implement UserDetailsService

You implement a custom `UserDetailsService` for more flexibility when loading user data.

1. Comment the previously created `JdbcUserDetailsManager` bean definition.
2. Open the `CustomUserDetailsService` class and implement the `UserDetailsService` interface.
3. This interface provides a `loadUserByUsername` method that is automatically called when a user needs to be loaded by `DaoAuthenticationProvider`.
4. Inject `AccountRepository` at the constructor level.
5. Implement the `loadUserByUsername` method.
   a. Look up `Account` by calling the `findByUsername` method on the repository passing the provided user name.

      The contract of the method expects `UserDetails` to be returned. Your `Account` is not of the `UserDetails` type.

   b. Wrap the account by returning `new AccountUserDetails(account)`.

      This custom class extends the `Account` class and implements the `UserDetails` interface.

   c. Throw a `UsernameNotFoundException` if the user was not found.
6. Register the class as a Spring-managed bean.

   Choose one of these two options:

   - Annotate the class with `@Component`.
   - Explicitly declare the class in the `SecurityConfig` configuration class.

   `DaoAuthenticationProvider` is configured with your `UserDetailsService`.
6. Run the application and verify that you can log in with keith as the user name and spring as the password.

## Task 7: Implement AuthenticationProvider

The `UserDetailsService` interface offers a simple contract to retrieve users, but it only has access to the user name to do the lookup. `AuthenticationProvider` offers access to the full `Authentication` object, which might be needed when access to the password is required.

1. Comment the previously created `UserDetailsService` bean definition.
2. Find the `CustomAuthenticationProvider` class in the `rewardsdining.security` package and implement the `authenticate` method.
   a. Obtain the user name from the `Authentication` object.
   b. Obtain the password from the `Authentication` object.
   c. Reuse the implemented `UserDetailsService` to load the user.
   d. Use the password encoder to verify that the provided password matches the user password.

      You can use the `match` method.
3. Register the class as a Spring-managed bean.

   Choose one of these two options:

   - Annotate the class with `@Component`.
   - Explicitly declare it in the `SecurityConfig` configuration class.
4. Run the application and verify that you can log in with keith as the user name and spring as the password.
5. Verify that the following line appears in the log.

```
TRACE: o.s.s.authentication.ProviderManager - Authenticating request with
CustomAuthenticationProvider (1/1)
```

## Task 8: Set Up an Embedded LDAP Server

You use an LDAP server to configure authentication.

1. Before configuring the LDAP server and authentication, verify that the Spring Security LDAP support and UnboundID are configured in `pom.xml`.
2. Inspect the `ldap-server.ldif` file in `src/main/resources`.

   The file contains the users that you will load into the LDAP server.
3. Add the following properties in `application.properties` to set up the embedded LDAP server and load users.

```
spring.ldap.embedded.base-dn=dc=rewardsdining,dc=org
spring.ldap.embedded.ldif=classpath:ldap-server.ldif
spring.ldap.embedded.port=8389
```

## Task 9: Set Up LdapAuthenticationProvider

The LDAP support does not use `UserDetailsService` because the LDAP bind authentication does not allow clients to read the password. Instead, you use `LdapAuthenticator`.

1. Comment the bean definition of the custom `AuthenticationProvider` that you implemented in an earlier task.
2. Uncomment the bean with `bindAuthenticator` name.

   This class uses LDAP bind authentication, where user credentials are sent to the LDAP server to perform authentication.
3. Uncomment the bean with the `ldapAuthenticationProvider` name.

   This is the LDAP provider that delegates to `BindAuthenticator` to perform authentication and to an authorities populator that populates user authorities based on the groups they belong to.
4. Run the application and log in with dollie as the user name and spring as the password.
5. Verify that `LdapAuthenticationProvider` is used in the log.

```
TRACE: o.s.s.authentication.ProviderManager - Authenticating request with Lda-
pAuthenticationProvider (1/1)
DEBUG: o.s.s.l.a.LdapAuthenticationProvider - Processing authentication request
for user: dollie
DEBUG: o.s.s.l.authentication.BindAuthenticator - Attempting to bind as
uid=dollie,ou=people,dc=rewardsdining,dc=org
```

## Task 10: Use Spring Boot Actuator to See Audit Events

Spring Boot Actuator enables an endpoint to see audit events. By default, it exposes authentication and authorization events.

1. Expose the endpoint by defining the property in `application.properties`.

   ```
   management.endpoints.web.exposure.include=auditevents
   ```

   By default, the endpoint is enabled but not exposed to HTTP.
2. Provide an in-memory repository by defining the following bean in the `SecurityConfig` class.

   Auditing requires a bean of the `AuditEventRepository` type in `ApplicationContext` to store the events.

```
@Bean
public InMemoryAuditEventRepository repository(){
    return new InMemoryAuditEventRepository();
}
```

3. Go to http://localhost:8080/actuator to verify that the endpoint is exposed and available.
4. Log in to the application with keith as the user name and spring as the password.
5. Log in to the application with dollie as the user name and spring as the password.

6. Log in to the application with a wrong combination of user name and password.
7. Go to http://localhost:8080/actuator/auditevents and verify that successful and unsuccessful authentications appear.

# Lab 4: Configuring Authorization

## Objective and Tasks

Protect HTTP endpoints with AccessDecisionsManager and AuthorizationManager:

1. Protect HTTP Endpoints with Role-Based Access Control
2. Define a Role Hierarchy
3. Protect Actuator Endpoints
4. Use Expressions for Authorization
5. Use the AuthorizationManager API
6. Bypass Security for the H2 Console
7. Inspect the Restaurant Entity
8. Create AuditorAware
9. Enable Spring Data JPA Auditing
10. Test Spring Data JPA Auditing

You must use the `40-authorization` project.

## Task 1: Protect HTTP Endpoints with Role-Based Access Control

The Reward Dining application defines different roles:

- USER: User using the Rewards Network application
- MANAGER: Restaurant manager
- ADMIN: Admin of the application

1. Protect the application endpoints by using the `authorizeRequests` method in the `SecurityFilterChain` definition.

   This step uses `AccessDecisionManager` and `AccessDecisionVoter`s to drive authorization.

   a. Allow GET on / for everyone.

   b. Allow GET on /restaurants (or any subresource) for all roles: USER, MANAGER, and ADMIN.

   c. Allow PUT on /restaurants/{id} for MANAGER and ADMIN.

   d. Allow GET on /accounts (or any subresource) for ADMIN.

   e. Require authentication for other endpoints.

   NOTE: Rules must be from more specific to less specific.

2. Access endpoints.
    a. Access http://localhost:8080 with no authentication.

       Access is granted.

    b. Access http://localhost:8080/restaurants with keith as the user name and spring as the password.

       Access is granted.

    c. Send a PUT request to update restaurant 1 with keith as the user name and spring as the password.

       curl -v -X PUT -H "Content-Type: application/json" -d '{"name":"AppleBeans", "location":"New York", "benefitPercentage":0.09}' -u keith:spring http://localhost:8080/restaurants/1

       Access is denied (403).

    d. Access http://localhost:8080/accounts with cornelia as the user name and spring as the password.

       Access is denied (403).

## Task 2: Define a Role Hierarchy

Roles have a clear hierarchy in the Rewards Dining application. However, all allowed roles must be specified when protecting HTTP endpoints.

1. Define a bean of type `RoleHierarchy`.
2. Create hierarchy:
    - ROLE_ADMIN > ROLE_MANAGER
    - ROLE_MANAGER > ROLE_USER
3. Protect GET to `/restaurants` with only the `USER` role.
4. Use `curl` to access `/restaurants` with the admin user chad.
5. Verify that you can access the restaurants endpoint.

   ```
   curl -v -u chad:spring http://localhost:8080/restaurants
   ```

## Task 3: Protect Actuator Endpoints

Spring Boot Actuator is enabled. For security purposes, all actuators other than `/health` are not exposed through HTTP by default.

1. Enable all actuator endpoints by defining a property in `application.properties`.

   ```
   management.endpoints.web.exposure.include=*
   ```

2. Run the application and access http://localhost:8080/actuator/env to verify that the endpoint is exposed.

   ```
   curl -v -u keith:spring http://localhost:8080/actuator/env
   ```

3. Protect all actuator endpoints except `/health` with the ADMIN role.

   Only admin users must have access to the actuator endpoints.

   You can use the `EndpointRequest.toAnyEndpoint().excluding("health")` request matcher.

4. Access the `/env` endpoint with keith as the user.

   A 403 response appears.

## Task 4: Use Expressions for Authorization

Sometimes role-based access control is not enough. You protected PUT requests to /restaurants/{id} with the role MANAGER or ADMIN. You verify that the user is the owner of the restaurant. All managers can update any restaurant.

1. Find the `RestaurantAuthorizer` class in the `rewardsdining.security` package.
2. Define a method with the signature.

   ```
   public boolean isOwner(Authentication authentication, long restaurantId)
   ```

3. Implement the method.
   a. Delegate to `restaurantManager` to retrieve the restaurant by id.
   b. Return true if the authorization name matches the restaurant owner user name.
4. Change the rule to restrict access to PUT on /restaurants/{id} and use the access method to define an expression in the form of SpEL.

   You must allow access to users with the ADMIN role or the MANAGER role only if they are owners of the restaurant.
5. Run the application and use curl to send PUT requests to `/restaurants/{id}`.
   a. Use keith as the user.

   ```
   curl -v -X PUT -H "Content-Type: application/json" -d '{"name":"AppleBeans",
   "location":"New York", "benefitPercentage":0.09}' -u keith:spring
   http://localhost:8080/restaurants/1
   ```

   A 403 response appears because keith has only the USER role.

   b. Use dollie as the user.

   ```
   curl -v -X PUT -H "Content-Type: application/json" -d '{"name":"AppleBeans",
   "location":"New York", "benefitPercentage":0.09}' -u dollie:spring
   http://localhost:8080/restaurants/1
   ```

   A 204 response appears because dollie is the owner of the restaurant.

   c. Use cornelia as the user.

   ```
   curl -v -X PUT -H "Content-Type: application/json" -d '{"name":"AppleBeans",
   "location":"New York", "benefitPercentage":0.09}' -u cornelia:spring
   http://localhost:8080/restaurants/1
   ```

   A 403 response appears because cornelia is a manager but not the owner.

   a. Use chad as the user.

   ```
   curl -v -X PUT -H "Content-Type: application/json" -d '{"name":"AppleBeans",
   "location":"New York", "benefitPercentage":0.09}' -u chad:spring
   http://localhost:8080/restaurants/1
   ```

   A 204 response appears because chad has the ADMIN role.

## Task 5: Use the AuthorizationManager API

The `AuthorizationManager` API was introduced in Spring Security 5.6. This simple API for authorization replaces `AccessDecisionManager` and `AccessDecisionVoters`.

1. Use the `authorizeHttpRequests` method instead of `authorizeRequests` in the `SecurityFilterChain` definition.

   Leave the matchers as they are. This step switches to the new `AuthorizationManager` API.

   An error appears in the `access` method. The `access` method does not expect an expression. It expects an `AuthorizationManager` instance.

2. Find the `RestaurantOwnerAuthorizationManager` class in the `rewardsdining.security` package and implement the `check` method.

   a. Return a `new AuthorizationDecision(true)` if the user has `ROLE_ADMIN` or if the user is the restaurant owner.

   b. If the authorization name is different from the owner user name, return `new AuthorizationDecision(false)`.

3. Run tests in the `RestaurantOwnerAuthorizationManagerTests` class under the `rewardsdining.security` test package.

   All tests should pass.

4. Use the `RestaurantOwnerAuthorizationManager` dependency in the `access` method to protect the PUT request to `/restaurant/{id}`.

   The `RestaurantOwnerAuthorizationManager` dependency was passed to `SecurityFilterChain`.

5. Run the application to update the restaurant using dollie as the user (the restaurant owner).

   ```
   curl -v -X PUT -H "Content-Type: application/json" -d '{"name":"AppleBeans",
   "location":"New York", "benefitPercentage":0.09}' -u dollie:spring
   http://localhost:8080/restaurants/1
   ```

6. Verify that a 204 response appears.

## Task 6: Bypass Security for the H2 Console

The Rewards Dining application uses H2, an embedded in-memory database. H2 provides a convenient web console to access the database that is autoconfigured by Spring Boot. For development purposes, it might be convenient to access the database. The H2 console might not need the same application security.

1. Identify the bean of type `WebSecurityCustomizer` in the `SecurityConfig` class.

   This bean allows ignoring security for specific paths. It is configured to bypass security for static resources such as `css` and `js` files. These resources do not go through the security filter chain.

2. Define `antMatcher` to ignore the `/h2-console/**` path.

3. Go to http://localhost:8080/h2-console to access the H2 console.

4. Change the JDBC URL to jdbc:h2:mem:rewards in the login form.

5. Inspect the database

## Task 7: Inspect the Restaurant Entity

Restaurants in the Reward Dining application define a percentage benefit to be awarded for eligible dining transactions. You track the restaurant entities that have auditing information about the user who modified them.

1. Open the `Restaurant` entity (in the common project).

   The `Restaurant` entity extends `AbstractAuditable<Account, Long>`, which is a Spring Data class that provides the `createdBy, createdDate, lastModifiedBy`, and `lastModifiedDate` fields. The `Restaurant` entity stores the information about the `Account` that has last modified it.

2. Verify that `Restaurant` class is annotated with `@EntityListeners(AuditingEntityListener.class)`.

   This class captures the auditing information when creating or editing the entity.

## Task 8: Create AuditorAware

To store the audit information, you must specify how Spring Data has to retrieve the current auditor. You provide a reference to `Account` that modifies the entity.

1. Create a `SpringSecurityAuditorAware` class under the `rewardsdining.security` package.
2. Implement the `AuditorAware<Account>` Spring Data interface.
3. Implement the `getCurrentAuditor` method by delegating to the static `getCurrentUser` method on `SecurityManager`.

   The `rewards-common` common project includes `SecurityManager`.

4. Annotate the class with `@Component` to make it a Spring-managed bean.

## Task 9: Enable Spring Data JPA Auditing

You have the entity with the auditing fields and `AuditorAware` to get the current user. You enable Spring Data JPA auditing.

1. Open the class annotated with `@SpringBootApplication`.
2. Annotate the class with `@EnableJpaAuditing`.

## Task 10: Test Spring Data JPA Auditing

You ensure that auditing works by implementing a test that modifies an existing `Restaurant` and verifies that the last modified field is updated with the current user.

1. Open the `RestaurantAuditorTests` test class.
2. In the `testRestaurantLastModifiedAuditor` method, change the benefit percentage of the restaurant to 20%.
3. Verify that the `lastModifiedBy Account` user name is equal to `TEST_ACCOUNT_USERNAME`.

# Lab 5: Method Security

## Objective and Tasks

Configure protecting methods:

1. Enable Method Security
2. Protect the Execution of a Method
3. Create a Custom Meta-Annotation

You must use the `41-method-security` project.

## Task 1: Enable Method Security

Method security uses AOP, and it is not enabled by default.

1. Enable method security by annotating the `SecurityConfig` class with `@EnableMethodSecurity`.

   This step enables method security by using the new `AuthorizationManager` API. `@Pre/PostAuthorize` and `@Pre/PostFilter` annotations will be processed.

## Task 2: Protect the Execution of a Method

In an earlier lab, you protected the PUT request to `/restaurants/{id}` not only with roles but also with some custom access control logic. Method security allows applying authorization at the method level and gives you more control and granular authorization.

1. Open the `RestaurantManager` class and find the `save` method.
2. Use the `@PreAuthorize` annotation to perform a security check before executing the method with an expression.
   a. Allow the ADMIN role to execute the method
   b. Allow owners of the restaurant to execute the method.

      You can delegate to the `isOwner` method from the `RestaurantAuthorizer`.
3. Run the tests in the `RestaurantManagerTests` class.

   All tests should all pass.

## Task 3: Create a Custom Meta-Annotation

Expressions in security annotations can become complex. If you need to reuse the expression, encapsulate the expression in a meta-annotation.

1. Create an annotation, `@IsOwner`, in the `rewardsdining.security` package.
   a. Annotate the `@IsOwner` annotation with `@PreAuthorize` and the expression that you defined in the previous task.
   b. Restrict the annotation to be used at the method or type level.

      You can use `@Target({ ElementType.METHOD, ElementType.TYPE })`.
   c. Set the retention to runtime with `@Retention(RetentionPolicy.RUNTIME)`.
2. Remove the `@PreAuthorize` annotation from the `save` method in `RestaurantManager` and use the `@IsOwner` as the new annotation.
3. Run the tests in the `RestaurantManagerTests` class again.

   All tests should pass.

# Lab 6: Security Testing

## Objective and Tasks

Test method and HTTP endpoints protection:

1. Simplify Method Security Tests with @WithMockUser
2. Create a Meta-Annotation to Test Admin
3. Use UserDetailsService Implementation
4. Use WithSecurityContextFactory
5. Test Form Login with MockMvc
6. Test Basic Authentication with MockMvc
7. Test the Protected Endpoints with MockMvc

You must use the `50-security-testing` project.

## Task 1: Simplify Method Security Tests with @WithMockUser

In the previous labs, you created `SecurityContext` explicitly before test methods.

You simplify the tests with the `@WithMockUser` annotation.

1. Find the `RestaurantManagerTests` class in the `rewardsdining.restaurant` test package.
2. Add the `@WithMockUser` annotation with the required user name and password in all test methods.
3. Remove the method call to initialize `SecurityContext` from all test methods.
4. Remove the `@AfterEach` method that clears `SecurityContext`.
5. Run the tests.

   All tests must pass.

## Task 2: Create a Meta-Annotation to Test Admin

Testing with the same user in different tests is common. You create a custom annotation to avoid specifying the same attributes in all tests and make the test reusable.

1. Create an annotation called `@IsAdmin`.
2. Annotate the `@IsAdmin` annotation with `@Target({ ElementType.METHOD, ElementType.TYPE })` and `@Retention(RetentionPolicy.RUNTIME)`.
3. Annotate it with `@WithMockUser(username = "admin", authorities = "ROLE_ADMIN")`.
4. Replace the `@WithMockUser` annotation on the `testAdminShouldBeAllowedToUpdate` method with your custom annotation.
5. Run the test.

   The test must pass.

## Task 3: Use UserDetailsService Implementation

With the `@WithMockUser` annotation, the specified user does not need to exist. You use `@WithUserDetails` to load the user with the existing `UserDetailsService` implementation in `ApplicationContext`.

1. Replace the annotation on the `testOwnerShouldBeAllowToUpdate` method with `@WithUserDetails("dollie")`.

   This annotation delegates to `UserDetailsService` to load the user. The user must exist.

2. Run the tests again.

   The tests must pass.

## Task 4: Use WithSecurityContextFactory

For more flexibility, the `@WithSecurityContext` annotation allows delegating to a factory class to create `SecurityContext`.

1. Create the `@WithMockCustomUser` annotation.
2. Annotate the `@WithMockCustomUser` annotation with `@Target({ ElementType.METHOD, ElementType.TYPE })` and `@Retention(RetentionPolicy.RUNTIME)`.
3. Annotate it with `@WithSecurityContext` and define the factory class to be `CustomSecurityContextFactory.class`.
4. Create a `CustomSecurityContextFactory` class that implements `WithSecurityContextFactory<WithMockCustomUser>`.
5. Implement the `createSecurityContext` method and return `SecurityContext` with your custom authentication object.
6. Set a `TestingAuthenticationToken` instance in the `SecurityContext` with the role USER_ROLE.
7. Replace the annotation on the `testUserShouldNotBeAllowToUpdate` method with `@WithMockCustomUser`.
8. Run the tests again.

   The tests must pass.

## Task 5: Test Form Login with MockMvc

The Rewards Dining application is configured to work with login authentication. You implement a test to verify that the application is configured correctly.

1. Find the `MvcSecurityTests` class in the `rewardsdining.security` test package.
2. Annotate the class with `@AutoConfigureMockMvc` to autoconfigure the `MockMvc` object with security filters.
3. Create a `testSuccessfulFormLogin` method and annotate it with `@Test`.
4. Use the `mockMvc` object.
   a. Perform `formLogin` with keith as the user name and spring as the password.

      You can use `formLogin() SecurityMockMvcRequestBuilder`.

   b. Expect the user is authenticated with the USER role.

      You can use `authenticated() SecurityMockMvcResultMatcher`.

5. Run the test.

   The test must pass

## Task 6: Test Basic Authentication with MockMvc

The Rewards Dining application is configured to also support basic authentication. You implement a test to verify that the application is configured correctly.

1. Find the `MvcSecurityTests` class in the `rewardsdining.security` test package.
2. Create a `testSuccessfulBasicAuthentication` method and annotate it with `@Test`.
3. Use the `mockMvc` object.
   a. Perform a get request to /.
   b. Add basic authentication headers by using `RequestPostProcessor httpBasic` with keith as the user name and spring as the password.
   c. Expect the user is authenticated with the USER role.

      You can use `authenticated() SecurityMockMvcResultMatcher`.
4. Run the test.

   The test must pass

## Task 7: Test the Protected Endpoints with MockMvc

You tested method protection. You implement a test to verify the correct security configuration for your application endpoints.

1. Find the `RewardsControllerTests` class in the `rewardsdining.reward.web` test package.
2. Annotate the class with `@AutoConfigureMockMvc` to autoconfigure the `MockMvc` object with security filters.
3. Run the test.

   The test fails because the /rewards endpoint is protected with ROLE_ADMIN.
4. Modify the test to use admin as the user name with the ADMIN role.

   You can use the `user()RequestPostProcessor`.
5. Run the test again.

   The test must pass.

# Lab 7: Managing Passwords

## Objective and Tasks

Store passwords securely and upgrade them for better security:

1. Inspect the Current Passwords
2. Find the Optimal BCrypt Strength
3. Change the BCryptPasswordEncoder Strength
4. Implement UserDetailsPasswordService
5. Upgrade Legacy Hashes

You must use the `60-password-handling` project.

## Task 1: Inspect the Current Passwords

The Rewards Dining application uses H2, an embedded in-memory database. H2 provides a web console to access the database, which is autoconfigured by Spring Boot.

1. Go to http://localhost:8080/h2-console to access the H2 console.

   You must change the JDBC URL to jdbc:h2:mem:rewards in the login form.

2. Check the password format for different users in the `T_ACCOUNT` table.

   Q1: Which hashing algorithms are currently used?

   A1: BCrypt and MD5.

## Task 2: Find the Optimal BCrypt Strength

The strength parameter in BCrypt determines the number of iterations that are performed for each password ($2^{\text{strength}}$ iterations). Increasing the strength makes calculating the hash computationally more expensive and difficult to crack.

In the Rewards Dining database, BCrypt is used to hash most passwords with a default strength of 10. Ideally, passwords must take between 0.5 and 1 second to hash. A `BCryptStrengthTester` class is provided in the `rewardsdining.security` package. It calculates the time needed to hash a password using different strengths.

1. Open the `BCryptStrengthTester` class in the `rewardsdining.security` package and review the implementation.
2. Create `CommnadLineRunner` that executes the static `startTest` method of `BCryptStrengthTester`.
3. Run the application.

   The logs display the milliseconds needed for different strengths.

   Q2: Which strength takes between 0.5 and 1 second?

   A2: Answers vary depending on your hardware.

## Task 3: Change the BCryptPasswordEncoder Strength

Depending on the hardware that your application runs on, 10 might be a weak work factor for BCrypt. This application database has passwords encoded in different algorithms. `DelegatingPasswordEncoder` supports different hashing algorithms, but the default strength for BCrypt is 10.

1. Replace the existing `PasswordEncoder` definition in the `SecurityConfig` class by a `DelegatingPasswordEncoder` instance that supports BCrypt and MD5.

   You must change the BCrypt strength to a value that makes hashing close to 1 second. Use the optimal strength given by `BCryptStrengthTester` in the previous task.

```
@Bean
public PasswordEncoder passwordEncoder() {
    String encodingId = "bcrypt";
    Map<String, PasswordEncoder> encoders = new HashMap<>();
    encoders.put(encodingId, new BCryptPasswordEncoder(<your-strength>));
    encoders.put("MD5", new MessageDigestPasswordEncoder("MD5"));
    return new DelegatingPasswordEncoder(encodingId, encoders);
}
```

   `DelegatingPasswordEncoder` uses deprecated password encoders to support legacy hashes.

## Task 4: Implement UserDetailsPasswordService

You increased the BCrypt strength, but the passwords are still hashed with the default strength and MD5 in the database. Only new accounts will use the new `PasswordEncoder`. You upgrade the existing password gradually.

1. Find the `UpgradePasswordService` class in `rewardsdining.security`.
2. Implement the `UserDetailsPasswordService` interface.

   This step provides a `updatePassword` method that is automatically called when a password must be upgraded.
3. Inject `AccountRepository` and `PasswordEncoder` at the constructor level.
4. Implement the `updatePassword` method.
   a. Find `Account` by calling the `findByUsername` method on the repository passing the provided `UserDetail` user name.
   b. Use `PasswordEncoder` to encode the new password and set it to the account.
   c. Use the repository to save the account with the new password.

      The contract of the method expects `UserDetail` to be returned. The Rewards Dining `Account` is not of the `UserDetail` type.
   d. Wrap the account by returning a `new AccountUserDetails(account)`.
5. Register the class as a Spring-managed bean.

   Choose one of these two options:

   - Annotate the class with `@Component`.
   - Explicitly declare the class in the `SecurityConfig` configuration class.

   The DaoAuthenticationProvider that is used is configured with your `UserDetailsPasswordService`.

## Task 5: Upgrade Legacy Hashes

When a user authenticates, the password encoder verifies if the stored password must be updated for better security and delegates to `UserDetailsPasswordService`. You authenticate with different users and examine how passwords are upgraded.

1. Inspect keith's password in the database.

   The `{bcrypt}$2a$10` prefix indicates that this password was hashed with Bcrypt, and its strength was set to 10.

2. Go to http://localhost:8080/login and log in as keith.

   - User name: keith
   - Password: spring

3. Check the password in the database for keith again.

   The password is updated with the `{bcrypt}$2a$13` prefix, which indicates that the password was upgraded using BCrypt with a strength of 13.

4. Check the password for cornelia.

   This password was hashed with MD5.

5. Go to http://localhost:8080/login and log in as cornelia.

   - User name: cornelia
   - Password: spring

6. Check the password in the database for cornelia again.

   The password is updated with the `{bcrypt}$2a$13` prefix, which indicates that the password was upgraded using BCrypt with a strength of 13.

# Lab 8: OAuth2 Social Login with GitHub

## Objective and Tasks

Use the OAuth2 Login feature:

1. Add the Required Dependencies
2. Create an OAuth App with GitHub
3. Generate a Client ID and a Secret
4. Configure the GitHub Login
5. Create a Custom OAuth2UserService
6. Manage Authentication Events

For this lab, you need a GitHub account.
You must use the `70-oauth2-login` project.

## Task 1: Add the Required Dependencies

The OAuth 2.0 Login feature enables application users to log in to the application with their existing account at an OAuth 2.0 or OpenID Connect 1.0 Provider.

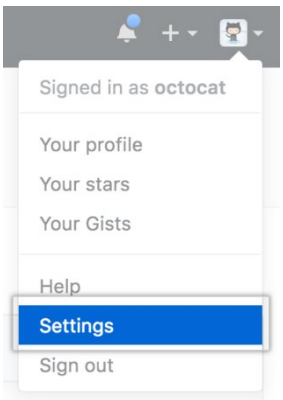
1. Add the started dependency in your `pom.xml` to use the Spring Security OAuth2/OpenID Connect client features.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

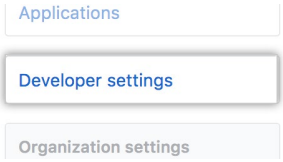
## Task 2: Create an OAuth App with GitHub

You must register a client to use an OAuth 2.0 provider such as GitHub.

1. Go to https://github.com/ and log in with your credentials.
2. Click **Settings** in the **Profile** menu.



2. Click **Developer settings** in the left sidebar.

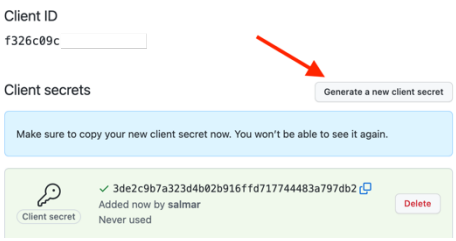



3. Click **OAuth Apps** and click **New OAuth App** or **Register a new application** to register your application.
4. Define the application properties.
   - Application name: Rewards Dining
   - Homepage URL: http://localhost:8080
   - Authorization callback URL: http://localhost:8080/login/oauth2/code/github
5. Register the application.

## Task 3: Generate a Client ID and a Secret

After creating the GitHub application, generate a client ID and a secret.

1. Access the registered application.
2. Click **Generate a new client secret** and store the client ID and the secret.




## Task 4: Configure the GitHub Login

For commonly used providers, such as Google, GitHub, Facebook, Okta, and so on, Spring Boot offers a client registration. This client registration is preconfigured with sensible default values such as the authorization and token URIs. The default values are available in the `CommonOAuth2Provider` enum. To configure a client registration for GitHub, you must provide the client ID and client secret.

1. Open `application.yaml` and add properties.

```
spring:
  security:
    oauth2:
      client:
        registration:
          github:
            client-id: <YOUR-CLIENT-ID>
            client-secret: <YOUR-CLIENT-SECRET>
```

2. In `SecurityConfig`, add a line in `SecurityFilterChain` to configure OAuth2 login along with form authentication.

```
.oauth2Login(withDefaults());
```

3. Run the application and go to http://localhost:8080/restaurants.

   You are redirected to the login page. Both form-based authentication and GitHub authentication are available.

3. Click the **GitHub OAuth** authentication and access with your credentials.

4. When you are redirected to `/restaurants`, verify your user name by clicking the user icon in the upper-right corner.

   Q1: What is your user name?

   A1: The user name is set to the user attribute id obtained from GitHub.

5. Check the logs and find the authentication object stored in `SecurityContext`.

```
DEBUG: o.s.s.w.c.HttpSessionSecurityContextRepository - Stored SecurityContex-
tImpl [Authentication=OAuth2AuthenticationToken
[Principal=Name: [122109], Granted Authorities: [[ROLE_USER, SCOPE_read:user]],
User Attributes: [{login=salmar, id=122109, node_id=MDQ6VXNlcjEyMjEwOQ…
```

   Q2: What is the type of the Authentication object?

   A2: OAuth2AuthenticationToken. When you used user name / password authentication, the type was UsernamePasswordAuthenticationToken. The principal type is DefaultOAuth2User.

6. Inspect the user attributes in the principal.

   The `login` attribute is the user name. By default, the `id` attribute is used as the principal name.

```
spring:
  security:
    oauth2:
      client:
        provider:
          github:
            user-name-attribute: login
```

7. In your `application.yaml`, override the provider configuration for GitHub and set the user name attribute to `login`.

8. Run the application and log in with GitHub.

   Q3: What is your principal name now?

   A3: It is the login user attribute.

## Task 5: Create a Custom OAuth2UserService

`UserDetailsService` was used to load user-specific data. When using OAuth2, the provider has the user details that can be retrieved from the UserInfo endpoint. This task is performed by `OAuth2UserService` and loads `OAuth2User` that will be available as the principal Authentication object.

1. Find the `CustomOAuth2UserService` class in the `rewardsdining.security` package.

   This class implements the OAuth2UserService interface.

2. Enhance the `load` method.

   The `load` method uses a delegator to load the user information from the provider after successful authentication.

   a. Use the account repository to find an account by user name (`oAuth2User.getName()`).

   b. If the account does not exist, create it in your records.

      A `createAccount` method is provided.

   c. Return `AccountUserDetails`.

      `AccountUserDetails` is your custom `Account` that also implements `OAuth2User`. The static `AccountUserDetails.from(account, oAuth2User)` method merges both.

3. Configure `oauth2Login` to use the new `OAuth2UserService` in the `SecurityFilterChain` definition by passing the `CustomOAuth2UserService` dependency as an argument in `SecurityFilterChain`.

```
.oauth2Login( oauth2 -> oauth2.userInfoEndpoint(
                userInfo -> userInfo.userService(customOAuth2UserService)));
```

4. Run the application and log in with your GitHub account.

   Q4: In the log, what is the type of the principal in `Authentication`?

   A4: The type is `AccountUserDetails`. It was `DefaultOAuth2User`.

## Task 6: Manage Authentication Events

Spring Security publishes authentication and authorization events. You create an event listener to update the last login date of a user.

1. Find the `AuthenticationEventListener` class under the `rewardsdining.security` package.
2. Create the `onSuccessfulAuthentication` method and add a parameter of the `InteractiveAuthenticationSuccessEvent` type.

   On successful authentications, the method is executed.
3. Annotate the method with `@EventListener`.
4. Implement the method.

   a. Use `accountRepository` to get an account by user name based on the authentication name.

   The authentication name is available in the event.

   b. Update the last login property and save the account.
5. Run the application and log in with your GitHub account.
6. Access the http://localhost:8080/h2-console H2 console and verify that the account was updated with the last login.
7. Log in with keith as the user name and spring as the password.
8. Verify that the last login column is updated.

# Lab 9: Configuring Spring Authorization Server

## Objective and Tasks

Use and configure Spring Authorization Server:

1. Create a Host Name for the Authorization Server
2. Configure the Provider Settings
3. Register a Client
4. Register Some Users
5. Use the Authorization Code Grant to Get a Token
6. Customize the Token with Authorities
7. Change the Token Expiration
8. Use the Client Credentials Grant to Get a Token

You must use the `80-authorization-server` project.

## Task 1: Create a Host Name for the Authorization Server

The Rewards Dining application and the authorization server run on your local machine. Both applications use session cookies. To avoid cookie overwrites, you ensure that the authorization server does not use localhost as the host name.

1. Open your `/etic/hosts` file.

   You can use `C:\Windows\System32\drivers\etc\hosts` if you are using Windows.

2. Add an entry.

   ```
   127.0.0.1 rewards-auth-server
   ```

## Task 2: Configure the Provider Settings

The authorization server allows you to define its configuration settings through the `ProviderSettings` class. The issuer identifier is one of the mandatory properties. It identifies the authorization server uniquely and allows clients to validate the issuer from authorization responses.

1. Find the `providerSettings` bean in the `AuthorizationServerConfig` class.
2. Set the issuer to http://rewards-auth-server:9090.

   You must always use the HTTPs scheme. In the lab environment, you use HTTP for simplicity.

3. Start the authorization server.

   OAuth2 and OpenID Connect define a discovery mechanism where the server publishes its metadata at a well-known URL.

4. Verify that the issuer is set correctly.
    a. Go to http://rewards-auth-server:9090/.well-known/oauth-authorization-server.
    b. Go to http://rewards-auth-server:9090/.well-known/openid-configuration.

## Task 3: Register a Client

The Rewards Dining application will use the authorization server, but it needs to be first registered as a client. Spring Authorization Server provides two implementations for the client repository: in-memory and JDBC. You use the JDBC client repository. H2 will be used as the in-memory embedded database.

1. Go to your `application.properties` and check the `spring.sql.init.schema-locations` property.

    This property creates the required schema in your database.

2. Find the `registeredClientRepository` bean and create a client for the Rewards Dining application.

    You can use the builder to define the properties.

    a. Set `clientId` to `rewards-dining`.

        This public identifier for the application is unique across all clients that the authorization server manages.

    b. Set `clientSecret` to `eH9A2N1BrjjsTPYUse79Orvez8HrST7r`.

        The secret is known only to the application and the authorization server. The secret must be random.

    c. Set the grant types that the client can use: `AuthorizationGrantType.AUTHORIZATION_CODE`, `AuthorizationGrantType.CLIENT_CREDENTIALS`, and `AuthorizationGrantType.REFRESH_TOKEN`.

        You can use the `authorizationGrantType` method. Each call adds the grant type to the list.

    d. Set `redirectUri` to http://localhost:8080/login/oauth2/code/rewards-dining.

        This URI will be used in redirect-based flows.

    e. Add `scope OidcScopes.OPENID`.

2. Add a second client to be used for testing.

    a. Set `clientId` to `rewards-debug`.

    b. Set `clientSecret` to `zH9A2N1BrjjsTPYUse79Orvez8HrST7r`.

    c. Set the `AuthorizationGrantType.AUTHORIZATION_CODE` grant type.

    d. Set `redirectUri` to https://oidcdebugger.com/debug.

3. Use the `save` method on `JdbcRegisteredClientRepository` to save the client.

## Task 4: Register Some Users

You registered the client. You add some users.

You explored several ways to obtain users in the previous labs.

1. Define a bean of type `UserDetailsService` in the `SpringSecurityConfiguration` class.
2. Create `InMemoryUserDetailsManager` with two users.
   - User name: keith, password: spring, and roles: ROLE_USER
   - User name: chad, password: spring, and roles: ROLE_USER, ROLE_ADMIN

## Task 5: Use the Authorization Code Grant to Get a Token

The OAuth 2.0 authorization framework defines four standard grant types: authorization code, implicit, resource owner password credentials, and client credentials. You use the authorization code grant to get an access token. Because you do not have any client application yet, you use https://oidcdebugger.com/ to test it.

1. In your browser, go to http://rewards-auth-server:9090/oauth2/authorize?client_id=rewards-debug&response_type=code&state=t0f3qf0330t8m9.
   - `rewards-debug` is the client that you previously configured.
   - `code` is the OAuth response type.
   - `t0f3qf0330t8m9` is a random value used by the client to maintain the state between the request and the callback. The state is used for preventing cross-site request forgery.
2. When you are redirected to the Authorization Server login form, use keith as the user name and spring as the password.

   After successful authentication, you are redirected to https://oidcdebugger.com/debug, where an authorization code is available. You must use the authorization code to exchange it with an access token. The authorization code is only valid for five minutes.
3. With the authorization code, make a POST request to the token endpoint for an access token.

```
curl --location --request POST 'http://rewards-auth-server:9090/oauth2/token' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=authorization_code' \
--data-urlencode 'client_id=rewards-debug' \
--data-urlencode 'client_secret=zH9A2N1BrjjsTPYUse79Orvez8HrST7r' \
--data-urlencode 'code=<YOUR-AUTHORIZATION-CODE>' \
--data-urlencode 'redirect_uri=https://oidcdebugger.com/debug'
```

   The authorization server validates the request and the client credentials and generates an access token.
4. Go to https://jwt.io/ and inspect the token.

   Q1: What is the expiration of the access token?

   A1: The token expires in 5 minutes. This value can be configured.

   Q2: Are roles included in the token?

   A2: No. Roles are not part of the token.

## Task 6: Customize the Token with Authorities

The access token does not contain the authorities that you defined when creating users in the authorization server. The authorization server offers an extension point to customize the OAuth 2.0 token attributes called `OAuth2TokenCustomizer`.

1. Find the `AuthoritiesTokenCustomizer` class in the `rewardsdining.auth.config` package.

   This class implements the `OAuth2TokenCustomizer` interface. You add a claim that contains the user authorities.

2. Implement the customize method.

   a. Get the current user authorities.

      You can use the provided `getAuthoritySet` method.

   b. Add a new claim named `authorities` with the user authorities.

      You can use the `claim` method from the `context.getClaims()` builder to add it.

3. Register the class as a Spring-managed bean.

   Choose one of these two options:

   - Annotate the class with `@Component`.
   - Explicitly declare it in the `AuthorizationServerConfig` configuration class.

4. Perform the steps in the previous task to retrieve a new access token with the user chad.

   Q3: Does the token contain authorities?

   A3: The token has a new claim with authorities.

```
{
  "sub": "chad",
  "aud": "rewards-debug",
  "nbf": 1636364246,
  "iss": "http://rewards-auth-server:9090",
  "exp": 1636364546,
  "iat": 1636364246,
  "authorities": [
    "ROLE_USER",
    "ROLE_ADMIN"
  ]
}
```

## Task 7: Change the Token Expiration

When registering a client, the `RegisteredClient` builder allows customizing some token properties, such as the access token expiration, with the `tokenSettings` method. This customization configures the token settings for all tokens issued for a client. You configure a different expiration time for access tokens depending on the user authorities. Admin users must have an expiration time of 15 minutes. Other users must have a 60-minute expiration time.

1. In `OAuth2TokenCustomizer`, open the `AuthoritiesTokenCustomizer` class to add the customization.
2. After setting the authorities claim, use the `expiresAt` method from the `context.getClaims()` builder to set the expiration time to 15 minutes for users with the role ADMIN and 60 minutes for all other users.

   A `calculateTokenExpiration` method is provided.
3. Create a token for chad (admin) and keith (user).

   You can use the steps in an earlier task.
4. Inspect the tokens.

   Different expiration times appear.

   NOTE: Log out of the authorization server at http://rewards-auth-server:9090/logout before using a new user.

## Task 8: Use the Client Credentials Grant to Get a Token

After seeing the authorization code grant, explore the client credentials grant. Use the `rewards-dining` client.

You do not need a browser.

1. Send a POST request to the token endpoint with three parameters.
   - `client_id`: Id of the client registered with the authorization server
   - `client_secret`
   - `grant_type` must be client_credentials for the client credentials grant.

```
curl --location --request POST 'http://rewards-auth-server:9090/oauth2/token' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'client_id=rewards-dining' \
--data-urlencode 'client_secret=eH9A2N1BrjjsTPYUse79Orvez8HrST7r' \
--data-urlencode 'grant_type=client_credentials'
```

   The authorization server validates the request and the client credentials and generates an access token.
2. Go to https://jwt.io/ and inspect the token.

# Lab 10: Using a Resource Server and Implementing an OAuth2 Client

## Objective and Tasks

Use the OAuth2 Resource Server and OAuth2 Client:

1. Configure the Resource Server
2. Create a Host Name for the Dashboard Application
3. Add a New Client in the Authorization Server
4. Configure the OAuth2 Client
5. Start the Application
6. Access the User Information
7. Validate the Access Token
8. Revoke the Token
9. Validate the Token Against the Authorization Server
10. Implement OpaqueTokenIntrospector

The Rewards Dining application has an endpoint to retrieve all rewards from the network. The endpoint can be used by admin users to see the network activity. A new application called Rewards Dashboard was created to provide a graphical representation of the rewards generated at every restaurant.

You use three projects in this lab:

- Resource Server (`90-resource-server`): Runs on port 8080
- Client (`90-reward-dashboard`): Runs on port 8082
- Authorization Server (`80-authorization-server-solution`): Runs on port 8081
  You use this solution from a previous lab.

## Task 1: Configure the Resource Server

The Rewards Dining application is the resource server. `RewardsRestController` (in the common project) provides an endpoint to retrieve all rewards.

1. In the `SecurityConfig` class, protect the `/rewards` endpoint with the admin role.
2. Configure the OAuth 2.0 Resource Server support with JWT in `SecurityFilterChain`.

   You can use `.oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt)`.
3. Add a property in the `application.yaml` authorization server URI.

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://rewards-auth-server:9091
```

   The Resource Server needs to validate the incoming token with the Authorization Server public keys. This property discovers the `JWKS` endpoint to fetch the public keys and validates the `iss` claim.
4. In `SecurityConfig`, update the `jwtGrantedAuthoritiesConverter` bean to use the `authorities` claim name and an empty authority prefix.

   By default, the resource server gets the user authorities from the JWT `scope` claim and prefixes them with `SCOPE_`. The JWTs issued by the authorization server have an `authorities` claim containing the user roles.
5. Run the tests in `RewardsRestControllerTests`.

   All tests should pass.

## Task 2: Create a Host Name for the Dashboard Application

The OAuth 2.1 specification defines that the use of localhost in redirect URIs is not recommended. The authorization server does not support the use of localhost in the redirect URI and fails when validating it. You create a host name for the Reward Dashboard application.

1. Open your `/etc/hosts` file.
2. Add a new entry.

   ```
   127.0.0.1 rewards-dashboard.local
   ```

## Task 3: Add a New Client in the Authorization Server

The Reward Dashboard application must be registered as a client in the authorization server.

1. Open `AuthorizationServerConfig` and add a new client in `RegisteredClientRepository`.

```
   RegisteredClient dashboardClient = RegisteredClient.withId(UUID.randomUUID().toString())
        .clientId("rewards-dashboard")
        .clientSecret("aH9A2N1BrjjsTPYUse79Orvez8HrST7r")

        .clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_POST)

        .clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)

        .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
        .authorizationGrantType(AuthorizationGrantType.REFRESH_TOKEN)

        .authorizationGrantType(AuthorizationGrantType.CLIENT_CREDENTIALS)
        .redirectUri("http://rewards-dashboard.local:8082/login/oauth2/code/rewards-dining")
        .redirectUri("http://rewards-dashboard.local:8082/authorize/oauth2/code/rewards-
dining")
        .scope(OidcScopes.OPENID)
        .build();
```

2. Save the client in the repository.

## Task 4: Configure the OAuth2 Client

The Rewards Dashboard client application authenticates the user against the authorization server and delegates to the resource server to retrieve the rewards.

1. In the `SecurityConfig` class, configure `SecurityFilterChain` to use OAuth2 Login and enable support for an OAuth2 client.

   You can use the `90-reward-dashboard` project.

2. Set up the OAuth2 provider and the client registration in `application.yaml`.

   Add the following lines:

```
spring:
  security:
    oauth2:
      client:
        registration:
          rewards-dining:
            client-id: rewards-dashboard
            client-secret: aH9A2N1BrjjsTPYUse79Orvez8HrST7r
            scope: openid
        provider:
          rewards-dining:
            issuer-uri: http://rewards-auth-server:9091
```

   The Rewards Dashboard uses `WebClient` to fetch the rewards from the resource server. The request needs to send an access token to be authenticated.

2. Configure `WebClient` to send the token resolving it from the current authentication.

   You must set the default `defaultOAuth2AuthorizedClient` to true in `ExchangeFilterFunction` in the `WebClient` definition.

3. In `DashboardController`, configure `webClient` to fetch rewards from the resource server.

   You can use the `rewardsDiningBaseUrl`/rewards.

## Task 5: Start the Application

You access the dashboard to see the distribution of rewards among restaurants in a pie chart. You must log in as an admin to see the dashboard.

1. Start the authorization server (`80-authorization-server-solution`).
2. Start the resource server (`90-resource-server`).
3. Start the client application (`90-reward-dashboard`).
4. Go to http://rewards-dashboard.local:8082/.

   You are redirected to the authorization server login page.

5. Log in with keith as the user name and spring as the password.

   Keith is not an admin. If the authentication is successful, the resource server denies access to rewards. The `Oh no! Status code: 403` message appears.

6. Log out of the application.

   If you try to log in again, you are automatically logged in. Logging out only invalidates the session from the Reward Dashboard application but not from the authorization server.

7. Go to http://rewards-auth-server:9091/logout to log out of the authorization server.
8. Log in again with chad as the user name and spring as the password.

   The dashboard appears.

## Task 6: Access the User Information

After successfully authenticating with chad as the user, you access the user and token information.

1. Find the `userInfo` method in the `DashboardController` class.
2. Add parameters.
   - `@RegisteredOAuth2AuthorizedClient OAuth2AuthorizedClient authorizedClient`
   - `@AuthenticationPrincipal OidcUser oauth2User`
3. Add `username`, `idToken`, and `userAttributes` to the model from `oauth2User`.
4. Add `accessToken` and `refreshToken` to the model from `authorizedClient`.
5. Go to http://rewards-dashboard.local:8082/user-info and verify that all values appear.

## Task 7: Validate the Access Token

The authorization server has an endpoint to introspect the token. You use it to verify that the token is valid.

1. Copy the access token from the `/user-info` endpoint.
2. Use `curl` to inspect the token on the authorization server.

```
curl -v curl --location --request POST 'http://rewards-auth-server:9091/oauth2/introspect' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'client_id=rewards-dashboard' \
--data-urlencode 'client_secret=aH9A2N1BrjjsTPYUse79Orvez8HrST7r' \
--data-urlencode 'token=<YOUR-TOKEN>'
```

3. Verify that the token is valid and active.

## Task 8: Revoke the Token

An expiration time is defined for the token. But the token can be revoked before it reaches the expiration time. You revoke the token to deny access to the Reward Dashboard application.

1. Copy the access token from the `/user-info` endpoint.
2. Use `curl` to inspect the token on the authorization server.

```
curl -v curl --location --request POST 'http://rewards-auth-server:9091/oauth2/revoke' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'client_id=rewards-dashboard' \
--data-urlencode 'client_secret=aH9A2N1BrjjsTPYUse79Orvez8HrST7r' \
--data-urlencode 'token=<YOUR-TOKEN>'
```

3. Verify that a 200 status code appears from the request.
4. Go to http://rewards-dashboard.local:8082/ to access the dashboard.

   Q1: Can you still access the dashboard?

   A1: Yes. The resource server validates the token locally. Even if you revoke the token, the resource server does not validate the token against the authorization server. You must wait until the token expires.

## Task 9: Validate the Token Against the Authorization Server

Validating JWT tokens locally is fast, but you cannot detect if a token was revoked. For critical resources and when revocation is required, remote validation can be used. But remote validation adds latency to the validation.

1. Find the `SecurityConfig` class in the resource server (`90-resource-server project`).
2. Enable the opaque token support (instead of JWT) for remote token introspection by replacing `OAuth2ResourceServerConfigurer::jwt` with `OAuth2ResourceServerConfigurer::opaqueToken`.

   Introspection can also be used with JWTs. The opaque token support is token format agnostic.
3. Add the opaque token configuration in `application.yaml`.

```
spring:
  security:
    oauth2:
      resourceserver:
        opaque-token:
            introspection-uri: http://rewards-auth-server:9091/oauth2/introspect
            client-id: rewards-dining
            client-secret: eH9A2N1BrjjsTPYUse79Orvez8HrST7r
```

4. Restart the resource server.
5. Go to http://rewards-dashboard.local:8082/ to access the dashboard.

   Q2: Can you access the dashboard?

   A2: A 403 response status code appears. The resource server does not use the previously implemented jwtGrantedAuthoritiesConverter. The token is validated remotely and the resource server assigns the authorities based on the token scopes. The authorities claim is not parsed, and roles are not assigned to the Authentication object.

## Task 10: Implement OpaqueTokenIntrospector

You implement `OpaqueTokenIntrospector` to map the authorities claim to the authentication object.

1. Find the `JwtOpaqueTokenIntrospector` class in the `rewardsdining.security` package.
2. Inject `JwtGrantedAuthoritiesConverter`, which you previously implemented, and store it in a private field.
3. In the introspect method, use `jwtGrantedAuthoritiesConverter` to convert the JWT token into a collection of authorities.

   These authorities will be passed to create DefaultOAuth2AuthenticatedPrincipal.
4. Mark the class as `@Component` to make it a Spring-managed bean.
5. Restart the resource server.
6. Go to http://rewards-dashboard.local:8082/ to access the dashboard.

   You are logged in as chad.

   Q3: Can you access the dashboard?

   A3: Yes.

7. Revoke the token.

   You can use the steps in an earlier task.

8. Go to http://rewards-auth-server:9091/logout to log out of the authorization server.
9. Go to http://rewards-dashboard.local:8082/ to access the dashboard.

   Q3: Can you access the dashboard?

   A3: No, because your token is revoked.