OAuth2 - SSO - Angular - Spring Boot

Version 12.1 authored by Daniel X on 2025/05/15 22:37

Table of Contents

OAuth2 - SSO - Angular - Spring Boot **4**

1. OKTA Setup **5**

1. Registration 5

2. Create Users 5

3. Add Claims 5

Steps to Add email as a Custom Claim in Access Token 5

4. Create UserManagementApp Application 5

Step 1: Create 2 OIDC Web Applications in OKTA 5

Step 2: Get the Credentials 6

Step 3: Enable Refresh Token & PKCE 6

5. Create application CourseManagementApp 6

2. Implement User Management Application **7**

1. Backend Implementation 7

1.1 Overview 7

1.1.1 Authentication & Authorization 7

1.1.2 Components 7

1.2 user-config-server 7

1.3 user-eureka-server 8

1.4 user-gateway 9

1.5 user-profile-svc 11

MySQL: 12

1.6 user-admin-svc 12

1.7 Authentication Test 12

Issue 1: 12

Issue 2 13

Issue 3 13

Issue 4: 14

3. Implement Course Management Application **17**

1. Backend Implementation 17

1.1 Overview 17

1.1.1 Authentication & Authorization 17

1.1.2 Components 17

1.2 course-config-server 17

1.3 course-eureka-server 18

1.4 course-gateway 19

1.5 course-query-svc 22

MySQL: 24

.... 24

1.6 course-management-svc 24

1.7 course-registration-svc 24

4. Create shared library common-lib 25

1. Overview 25

2. Implementation 25

2.1 WebClientConfig 26

2.2 ProfileWebClient 26

2.3 Models 27

2.4 Security 27

2.4.1 Anotation 27

2.5 Exception Handling 27

2.6 Troubleshooting 27

2.6.1 Unable to find main class on "mvn clean install" 27

.... 28

3. Use common-lib in Microservices 28

3.1 How to Use common-lib in Your Microservices 28

3.2 Troubleshooting During Implementation 28

3.2.1 Could not autowire. No beans of 'ProfileWebClient 28

3.2.2 IllegalArgumentException - Not a managed type: class com.dxu.sso.common.model.Course 28

3.2.3 Sopped Loading Configurations from course-config-server 29

3.2.4 Aspect is not triggered 29

.... 30

OAuth2 - SSO - Angular - Spring Boot

- Attachments
- 1. OKTA Setup
- 2. Implement User Management Application
- 3. Implement Course Management Application
- 4. Create shared library common-lib

1. OKTA Setup

1. Registration

https://dev-50623690-admin.okta.com/

daniel.xu4@ontario.ca / Cxxxxxx123!

2. Create Users

zxuz111@gmail.com 2HS6YjhE68dg

admin1sso@mailinator.com Gpu8Cke7Y2P5

test1sso@mailinator.com / PHUh33x9fvK9

test2sso@mailinator.com / uY7n7KcF45eZ

3. Add Claims

Steps to Add email as a Custom Claim in Access Token

1. **Go to OKTA Admin Console**
2. Navigate to:
   Security → API → Authorization Servers
3. Click your server (e.g., default)
4. Go to the **Claims** tab
5. Click **"Add Claim"**

E.g. Add claim "email" with the following expression:

```
(appuser != null) ? appuser.email : ""
```

4. Create UserManagementApp Application

- Spring Boot handles login redirects to OKTA
- OKTA returns code → backend exchanges for access/refresh/id tokens
- Backend maintains authenticated session
- Angular never directly communicates with OKTA
- Backend enforces role-based authorization from your DB

Step 1: Create 2 OIDC Web Applications in OKTA

create two confidential client applications:

✅ App 1 — **User Management Application (Backend)**

✅ App 2 — **Courses Management Application (Backend)**

For each:

1. **Go to**: OKTA Admin Console → Applications → Applications → **Create App Integration**
2. Choose:
    - **Sign-in method**: *OIDC - OpenID Connect*
    - **Application type**: *Web Application*
    - Click **Next**
3. Fill in details:
    - **Name**: e.g., UserManagementApp
    - **Grant type**:
        - ✅ Authorization Code
        - ✅ Refresh Token (checked by default for Web App)
        - ✅ Client Credentials (for microservice-to-microservice calls)
    - **Sign-in redirect URIs**: http://localhost:9001/login/oauth2/code/okta (on gateway)
    - **Sign-out redirect URIs**: http://localhost:9001/logout (on gateway)
4. Click **Save**

Step 2: Get the Credentials

After saving, you will get:

- **Client ID**
- **Client Secret**
- **Issuer URL**: https://dev-50623690.okta.com/oauth2/default
    - Navigate to: **Security** → **API** → **Authorization Servers**

You'll need these for Spring Boot's application.yml.

Step 3: Enable Refresh Token & PKCE

- **PKCE is required** and automatically enabled for confidential clients in Spring Security when the user-agent is a browser.
- **Refresh Token** is enabled by default for Web Applications in OKTA.

5. Create application CourseManagementApp

Follow the same steps. Just use port **9002** which is the gateway of CourseManagementApp

2. Implement User Management Application

1. Backend Implementation

1.1 Overview

1.1.1 Authentication & Authorization

- **OAuth2 login via OKTA** is handled **only by user-gateway**
- Each backend service is:
  - **Stateless**
  - Secured via **JWT** (relayed by gateway using TokenRelay)
  - Uses spring-boot-starter-oauth2-resource-server for validating tokens
- **Authorization** is managed by application itself

1.1.2 Components

Component	Purpose	Port
user-gateway	Central entry point (handles login & token relay)	9001
user-profile-svc	Saves profile of the logged-in OKTA user	9002
user-admin-svc	Allows admin to manage user accounts	9003
config-server	Centralized configuration	9004
eureka-server	Service discovery	**9761**
MySQL DB 8	Stores internal user profile info (email, roles, etc.)	—

Spring Boot: 3.4.5

Spring Cloud: 2024.0.1

1.2 user-config-server

Dependencies:

- spring-cloud-config-server
- spring-boot-starter-actuator

application.yml

```yaml
server:
  port: 9004

spring:
  application:
    name: user-config-server
  profiles:
    active: native

  cloud:
    config:
      server:
        native:
          search-locations: file:../user-config-repo
      fail-fast: true
```

@EnableConfigServer

Add the following YAML files for the applications that want to externalize their configurations

- **user-eureka-server**.yml
- **user-gateway**.yml
- **user-profile-svc**.yml
- **user-admin-svc**.yml

1.3 user-eureka-server

Dependencies:

- spring-cloud-starter-netflix-eureka-server
- spring-cloud-starter-config
- spring-boot-starter-actuator

application.yml

```yaml
server:
  port: 9761

spring:
  application:
    name: user-eureka-server
  config:
    import: configserver:http://localhost:9004

eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
    service-url:
      defaultZone: http://localhost:9761/eureka/

  server:
    enable-self-preservation: false
    enable-replication: false
```

@EnableEurekaServer

Note: The following configurations are MUST to avoid replication which cause issues:

```
eureka:
  client:
    service-url:
      defaultZone: http://localhost:9761/eureka/

  server:
    enable-self-preservation: false
    enable-replication: false
```

Spring Cloud Eureka Server is a **special case** where most of its critical configuration **must remain local**, and **cannot be externalized** to the Config Server, because:

- **Startup Dependency Loop**
  - The Eureka Server needs its config **before** it can connect to other services.
  - If you try to externalize application.yml to the config server, but the Eureka server itself hasn't started yet, it **cannot connect to the config server**.
  - This causes a **bootstrap deadlock**.
- **It’s not a config client by default**
  - Eureka Server **serves** the registry and is expected to be **fully self-contained**.

1.4 user-gateway

Dependencies:

- spring-cloud-starter-gateway
- spring-boot-starter-oauth2-client
- spring-boot-starter-security
- spring-cloud-starter-config
- spring-cloud-starter-netflix-eureka-client
- spring-boot-starter-actuator

application.yml

```yaml
server:
  port: 9001

spring:
  application:
    name: user-gateway
  config:
    import: configserver:http://localhost:9004

  cloud:
    gateway:
      routes:
        - id: user-profile
          uri: lb://user-profile-svc
          predicates:
            - Path=/api/profile/**
          filters:
            - TokenRelay

        - id: user-admin
          uri: lb://user-admin-svc
          predicates:
            - Path=/api/admin/**
          filters:
            - TokenRelay

  security:
    oauth2:
      client:
        registration:
          okta:
            client-id: 0oaooqy915xbkRCKT5d7
            client-secret: Ew-4IiHuFr6cgY1w3JhJ5PFWzvB-yZGx6xdD_y3rxhSWioOG-U_NGsCViCvlAxfe
            scope: openid, profile, email
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/okta"
        provider:
          okta:
            issuer-uri: https://dev-50623690.okta.com/oauth2/default

eureka:
  client:
    service-url:
      defaultZone: http://localhost:9761/eureka/

  instance:
    prefer-ip-address: true
    hostname: localhost
```

Note: It turns out the externalized configuration is not working for user-gateway as well. Have to move all configurations to application.yml

With "**issuer-uri: https://dev-50623690.okta.com/oauth2/default**",

Spring will:

- Fetch public signing keys from OKTA:

  https://dev-50623690.okta.com/oauth2/default/v1/keys

- Use those to validate:
  - Signature
  - Expiry (exp)
  - Issuer (iss)
  - Audience (aud) if configured

✅ So if the token is invalid, the request is rejected with **401 Unauthorized before it hits your controller**.

1.5 user-profile-svc

Dependencies:

- spring-boot-starter-web
- spring-boot-starter-data-jpa
- spring-boot-starter-oauth2-resource-server
- spring-boot-starter-oauth2-client
- spring-boot-starter-security
- spring-cloud-starter-config
- spring-cloud-starter-netflix-eureka-client
- mysql-connector-j
- lombok

application.yml

```yaml
spring:
  application:
    name: user-profile-svc

  config:
    import: configserver:http://localhost:9004

# See other configurations in user-config-repo/user-profile-svc.yml
```

MySQL:

```
CREATE DATABASE ssouser CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;
CREATE USER 'ssouser'@'localhost' IDENTIFIED BY 'passw0rd';
GRANT ALL PRIVILEGES ON ssouser.* TO 'ssouser'@'localhost';
FLUSH PRIVILEGES;
```

1.6 user-admin-svc

Mostly same as user-profile-svc

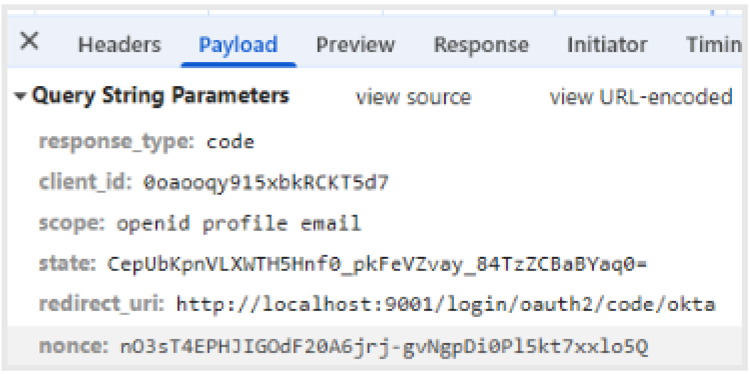
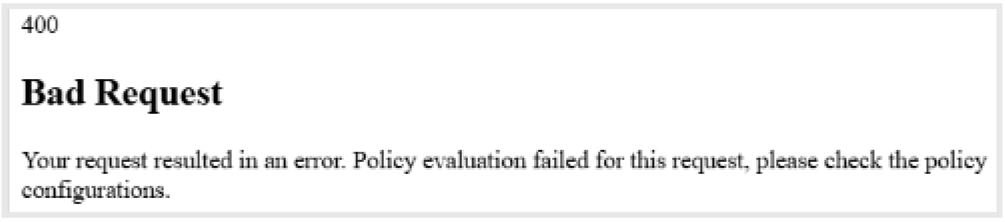
1.7 Authentication Test

Go to http://localhost:**9001**/api/profile/me

Issue 1:

Expected: Redirect to OKTA login page

Actual:

http://localhost:9001/api/profile/me	302. Location: /oauth2/authorization/okta
http://localhost:9001/oauth2/authorization/okta	302. Location: https://dev-50623690.okta.com/oauth2/default/v1/authorize?response_type=code&client_id=0oaooqy915xbkRCKT5d7&scope=openid%20profile%20email&state=CepUbKpnVLXWTH5Hnf0_pkFeVZvay_84TzZCBaBYaq0%3D&redirect_uri=http://localhost:9001/login/oauth2/code/okta&nonce=nO3sT4EPHJIGOdF20A6jrj-gvNgpDi0Pl5kt7xxlo5Q
The above URL of the 302 location	 Query String Parameters response_type: code client_id: 0oaooqy915xbkRCKT5d7 scope: openid profile email state: CepUbKpnVLXWTH5Hnf0_pkFeVZvay_84TzZCBaBYaq0= redirect_uri: http://localhost:9001/login/oauth2/code/okta nonce: nO3sT4EPHJIGOdF20A6jrj-gvNgpDi0Pl5kt7xxlo5Q Headers Payload Preview Response Initiator Timin view source view URL-encoded  400 Bad Request Your request resulted in an error. Policy evaluation failed for this request, please check the policy configurations.

Solution: Create an Access Policy + Rule

✅ Step 1: Go to Authorization Server Settings

1. Open your OKTA Admin Console
2. Go to:
   Security → API → Authorization Servers
3. Click on the **default** authorization server

✅ Step 2: Create a New Access Policy

1. Go to the **Access Policies** tab
2. Click **“Add Policy”**
3. Fill in:
   - **Name**: Allow Spring Boot Clients
   - **Description**: Policy to allow authorization code flow for user-gateway
   - Leave defaults for user matching
4. Click **“Create Policy”**

✅ Step 3: Add a Rule to That Policy

After creating the policy:

1. Click **“Add Rule”**
2. Fill in:
   - **Name**: Allow code flow for gateway
   - **IF Grant type is**: ✅ Authorization Code, ✅ Refresh Token
   - **IF User is**: ✅ Any user
   - **IF Client is**: ✅ (select your app client — user-gateway / client ID: 0oaooqy915xbkRCKT5d7)
3. Click **Create Rule**

Issue 2

Error: **UnknownHostException** for dxpsdesktop.mshome.net

Solution:

For user-profile-svc, user-admin-svc and user-gateway, add the following configuration:

```yaml
eureka:
  instance:
    prefer-ip-address: true
    hostname: localhost
```

Issue 3

Error:

In browser	**URL**: https://dev-50623690.okta.com/oauth2/default/v1/authorize?response_type=code&client_id=0aooqy915xbkRCKT5d7 &scope=openid%20profile%20email&state=LyJq5kOzBNHCfBKGrKy6zNLM7KFXO9_KZ2KYhhvaBtM%3D &redirect_uri=http://172.21.48.1:9002/login/oauth2/code/okta&nonce=HEqMZvTveDA0yLLvHmVeVKdF0n0aRY_ZC5jibYO_H34 **Response**: 400 bad request Your request resulted in an error. The 'redirect_uri' parameter must be a Login redirect URI in the client app settings: https://dev-50623690-admin.okta.com/admin/app/oidc_client/instance/0aooqy915xbkRCKT5d7#tab-general

Solution:

Remove any Spring Boot login configs from **user-profile-svc** and **user-admin-svc**, which is:

```yaml
spring:
  security:
    oauth2:
      client:
        registration:
          okta:
            client-id: 0aooqy915xbkRCKT5d7
            client-secret: Ew-4IiHuFr6cgY1w3JhJ5PFWzvB-yZGx6xdD_y3rxhSWioOG-U_NGsCViCvlAxfe
            scope: openid, profile, email
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/okta"
        provider:
          okta:
            issuer-uri: https://dev-50623690.okta.com/oauth2/default
```

Issue 4:

Now, the log in page is displayed, but the URL is **http://172.21.48.1:9002/login**, 172.21.48.1 is my machine's IP address

Reason:

Spring Boot auto-configures security using default behavior because:

- You **included spring-boot-starter-security**
- But you **didn't override or disable** the default security configuration

When no explicit SecurityFilterChain bean is defined, Spring Security falls back to:

- Securing **all endpoints**
- Showing a default login page at /login

Solution:

Add a **SecurityConfig** class to

- rely on the **Gateway** to enforce auth for **request from UI**
- **permit all** requests of "**/public/****"
- **Other** requests must be **authenticated**

```
@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests(authorize -> authorize
                        .requestMatchers("/public/**").permitAll()
                        .anyRequest().authenticated())
                .csrf(csrf -> csrf.disable())
                .oauth2ResourceServer(oauth2 -> oauth2.jwt(Customizer.withDefaults()));

        return http.build();
    }
}
```

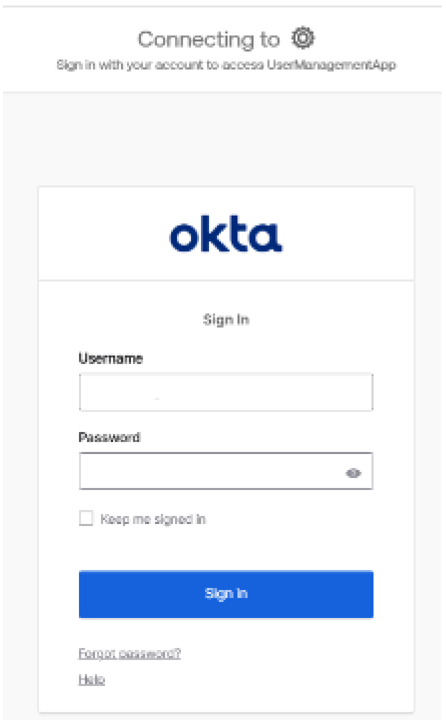
add the **Spring Security OAuth2 Resource Server** dependency

```
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
        </dependency>
```

Also, Spring Security needs a **JwtDecoder bean**

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://dev-50623690.okta.com/oauth2/default
```

Now access **http://localhost:9001/api/profile/me**, and the OKTA login page is displayed:



3. Implement Course Management Application

1. Backend Implementation

1.1 Overview

1.1.1 Authentication & Authorization

- **OAuth2 login via OKTA** is handled **only by course-gateway**
- Each backend service is:
    - **Stateless**
    - Secured via **JWT** (relayed by gateway using TokenRelay)
    - Uses spring-boot-starter-oauth2-resource-server for validating tokens
- **Authorization** is managed by application itself
- **User roles (student, teacher, admin)** are retrieved from the **User Management Application's API**

1.1.2 Components

Component	Port	Purpose
course-gateway	9011	Entry point; handles OKTA login
course-query-svc	9012	Read-only API for all roles
course-management-svc	9013	Admin/teacher APIs for course updates
course-registration-svc	9014	Student registration + teacher approval
eureka-server (shared)	9762	Service discovery
config-server (shared)	9015	Centralized config
MySQL (shared)	—	Stores course and registration data

1.2 course-config-server

Dependencies:

- spring-cloud-config-server
- spring-boot-starter-actuator

application.yml

```yaml
server:
  port: 9015

spring:
  application:
    name: course-config-server
  profiles:
    active: native

  cloud:
    config:
      server:
        native:
          search-locations: file:../course-config-repo
      fail-fast: true
```

@EnableConfigServer

Issue: Does not load configurations from folder "course-config-repo"
Reason: IntelliJ defaults to the **working directory of the first module with a main() method**, or sometimes the directory containing the .iml file for that module.
Solution:
Explicitly set **Working directory** for **course-config-server** in **Edit configuration**
    D:\workspace\sso-demo\sso-okta-app-authorization\backend\course-management-app\course-config-server

1.3 course-eureka-server

Dependencies:

- spring-cloud-starter-netflix-eureka-server
- spring-cloud-starter-config
- spring-boot-starter-actuator

application.yml

```yaml
server:
  port: 9761

spring:
  application:
    name: user-eureka-server
  config:
    import: configserver:http://localhost:9004

eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
    service-url:
      defaultZone: http://localhost:9761/eureka/

  server:
    enable-self-preservation: false
    enable-replication: false
```

@EnableEurekaServer

Note: The following configurations are MUST to avoid replication which cause issues:

```
eureka:
  client:
    service-url:
      defaultZone: http://localhost:9761/eureka/

  server:
    enable-self-preservation: false
    enable-replication: false
```

Spring Cloud Eureka Server is a **special case** where most of its critical configuration **must remain local**, and **cannot be externalized** to the Config Server, because:

- **Startup Dependency Loop**
  - The Eureka Server needs its config **before** it can connect to other services.
  - If you try to externalize application.yml to the config server, but the Eureka server itself hasn't started yet, it **cannot connect to the config server**.
  - This causes a **bootstrap deadlock**.
- **It's not a config client by default**
  - Eureka Server **serves** the registry and is expected to be **fully self-contained**.

1.4 course-gateway

Dependencies:

- spring-cloud-starter-gateway
- spring-boot-starter-oauth2-client
- spring-boot-starter-security

- spring-cloud-starter-config
- spring-cloud-starter-netflix-eureka-client
- spring-boot-starter-actuator

application.yml

```yaml
server:
  port: 9011

spring:
  application:
    name: course-gateway
  config:
    import: configserver:http://localhost:9015

  cloud:
    gateway:
      routes:
        - id: course-query
          uri: lb://course-query-svc
          predicates:
            - Path=/api/courses/**
          filters:
            - TokenRelay

        - id: course-management
          uri: lb://course-management-svc
          predicates:
            - Path=/api/course-mgmt/**
          filters:
            - TokenRelay

        - id: course-registration
          uri: lb://course-registration-svc
          predicates:
            - Path=/api/registration/**
          filters:
            - TokenRelay

  security:
    oauth2:
      client:
        registration:
          okta:
            client-id: 0oaoor5c1wl5NU1ef5d7
            client-secret: AC-_PThmy1t69J7Q1Je2g4HNRXq1mPqAUGDspKsy6dTFGk7RVV00a5jKiwZVuJ3z
            scope: openid, profile, email
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/okta"
        provider:
          okta:
            issuer-uri: https://dev-50623690.okta.com/oauth2/default

eureka:
  client:
    service-url:
      defaultZone: http://localhost:9762/eureka/

  instance:
    prefer-ip-address: true
    hostname: localhost
```

1.5 course-query-svc

Dependencies:

- spring-boot-starter-web
- spring-boot-starter-data-jpa
- spring-boot-starter-oauth2-resource-server
- spring-boot-starter-oauth2-client
- spring-boot-starter-security
- spring-cloud-starter-config
- spring-cloud-starter-netflix-eureka-client
- mysql-connector-j
- lombok

application.yml

```yaml
spring:
  application:
    name: course-query-svc

  config:
    import: configserver:http://localhost:9015

# See other configurations in course-config-repo/course-query-svc.yml
```

course-query-svc.yml

```yaml
server:
  port: 9012

spring:
  config:
    activate:
      on-profile: default

  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://dev-50623690.okta.com/oauth2/default

  datasource:
    url: jdbc:mysql://localhost:3306/ssocourse
    username: ssocourse
    password: passw0rd

  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true

eureka:
  client:
    service-url:
      defaultZone: http://localhost:9762/eureka/

  instance:
    prefer-ip-address: true
    hostname: localhost

# URL of User Profile API in User Management App
user-profile:
  url: http://localhost:9002/api/profile/me
```

As per best practice, the communication between backend microservices should not via gateway, so the CourseController is calling ProfileController directly (9002).

To protect these direct API calls, update "**anyRequest().permitAll()**" the **SecurityConfig** of **user-profile-svc** which is the application of the called API as below:

```java
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests(authorize -> authorize
                        .requestMatchers("/public/**").permitAll()
                        .anyRequest().authenticated())
                .csrf(csrf -> csrf.disable())
                .oauth2ResourceServer(oauth2 -> oauth2.jwt(Customizer.withDefaults()));

        return http.build();
    }
```

MySQL:

```
CREATE DATABASE ssocourse CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;
CREATE USER 'ssocourse'@'localhost' IDENTIFIED BY 'passw0rd';
GRANT ALL PRIVILEGES ON ssocourse.* TO 'ssocourse'@'localhost';
FLUSH PRIVILEGES;
```

1.6 course-management-svc

Mostly same as course-query-svc

1.7 course-registration-svc

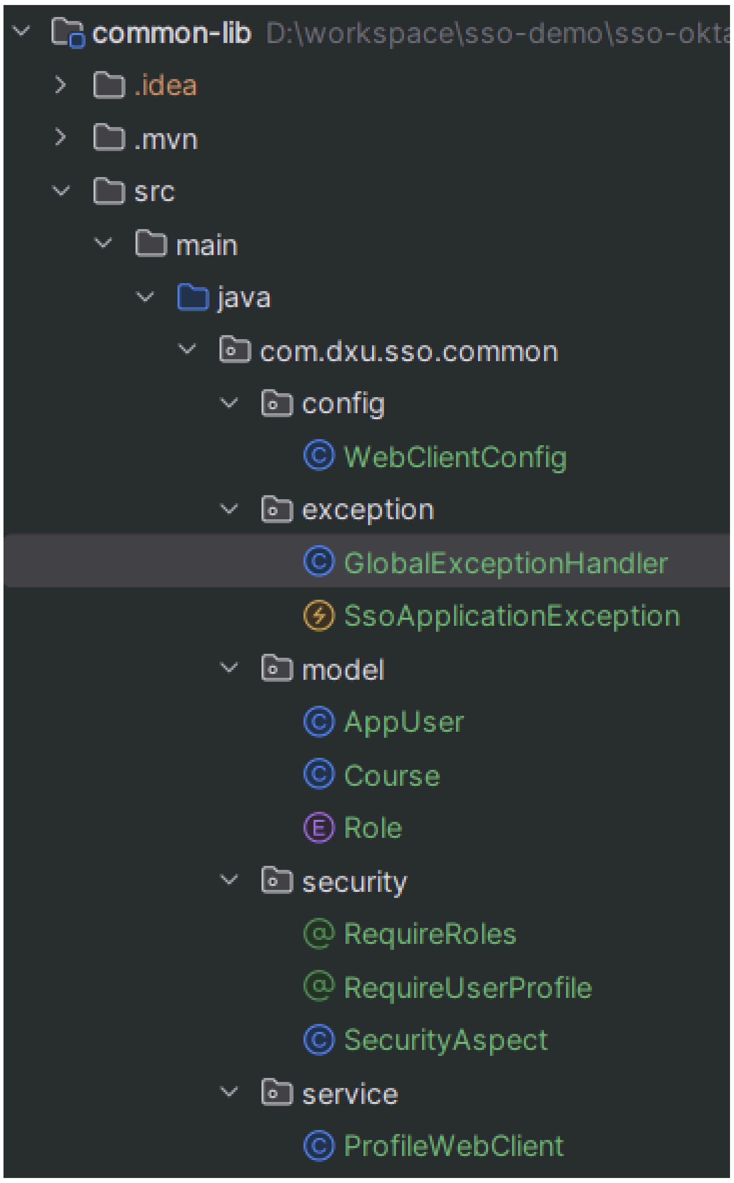
Mostly same as course-query-svc

4. Create shared library common-lib

1. Overview

For reusability and preventing duplicate code, move shared code to a shared module that can be reused by both the Course Management and User Management applications.

2. Implementation



```
common-lib D:\workspace\sso-demo\sso-okt
  .idea
  .mvn
  src
    main
      java
        com.dxu.sso.common
          config
            WebClientConfig
          exception
            GlobalExceptionHandler
            SsoApplicationException
          model
            AppUser
            Course
            Role
          security
            RequireRoles
            RequireUserProfile
            SecurityAspect
          service
            ProfileWebClient
```

- **WebClientConfig**: Create WebClient.builder that is shared by applications to do the inter-application commucation
- **GlobalExceptionHandler**: The global exception handler
- **model**: Shared POJO, entity, constants, etc
- **security**: Annotations for custom authentication/authorization shared by all applications
- **ProfileWebClient**: Shared service that contains shared operations such as fetch profile information from **/api/profile/me** in **user-profile-svc**

2.1 WebClientConfig

The consuming application use @Qualifier("commonWebClientBuilder") to distinguish with other WebClient (if exists)

```
    @Bean
    @Qualifier("commonWebClientBuilder")
    public WebClient.Builder commonWebClientBuilder() {
        return WebClient.builder();
    }
```

2.2 ProfileWebClient

Service of handling operations shared by applications. E.g.

```
    public AppUser getUserProfile() {
        log.info("Fetching user profile");

        ServletRequestAttributes attrs = (ServletRequestAttributes)
RequestContextHolder.getRequestAttributes();
        if (attrs == null) return null;

        String authHeader = attrs.getRequest().getHeader(HttpHeaders.AUTHORIZATION);
        if (authHeader == null || !authHeader.startsWith("Bearer ")) return null;

        return webClientBuilder.build()
                .get()
                .uri(userProfileUrl)
                .header(HttpHeaders.AUTHORIZATION, authHeader)
                .retrieve()
                .onStatus(status -> status.is4xxClientError(), response -> Mono.empty())
                .bodyToMono(AppUser.class)
                .block();
    }
```

Optimization - Cache the AppUser object per request using Spring's @RequestScope, to avoid multiple calls to /api/profile/me in the same request

```java
@RequestScope
@Component
@Getter
@Setter
public class UserContext {
    private AppUser appUser;
}

@Service
public class ProfileWebClient {
    ... ...
    public AppUser getUserProfile() {
        log.info("Fetching user profile");

        // Return cached user if already fetched
        if (userContext.getAppUser() != null) {
            return userContext.getAppUser();
        }

        ... ...
        AppUser user = ... ...;

        userContext.setAppUser(user); // ✅ cache it
        return user;
    }
```

2.3 Models

Shared POJO, entities, constants, etc.

2.4 Security

2.4.1 Anotation

- **@RequireRoles**: Ensures the user has one of the specified roles based on their profile from **/api/profile/me**. If not, throws a **403** ApplicationException
- **@RequireUserProfile**: Ensures the user is registered in the User Management application. If the profile is not found, throws a **403** ApplicationException

2.5 Exception Handling

- SsoApplicationException: The global exception handler shared by applications

2.6 Troubleshooting

2.6.1 Unable to find main class on "mvn clean install"

Reason: common-lib is a **shared library**, it does **not need a main() class**, and it **should not be packaged as a Spring Boot executable JAR**Type your error message here.
Solution: **Remove** the Spring Boot plugin:

3. Use common-lib in Microservices

3.1 How to Use common-lib in Your Microservices

- Add common-lib dependency in each microservice

```xml
<dependency>
    <groupId>com.dxu.sso.common.lib</groupId>
    <artifactId>common-lib</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

- Enable AOP in each microservice

```java
@SpringBootApplication
@EnableAspectJAutoProxy
public class CourseManagementServiceApplication {
    ...
}
```

3.2 Troubleshooting During Implementation

3.2.1 Could not autowire. No beans of 'ProfileWebClient

Reason: Spring **doesn't automatically scan components (@Service, @Component, etc.) in external libraries**

Solution

Explicit **@ComponentScan** in the consuming applicatio

```java
@ComponentScan(basePackages = {
        "com.dxu.sso.course.query",
        "com.dxu.sso.common"           // shared library "common-lib"
})
```

3.2.2 IllegalArgumentException - Not a managed type: class com.dxu.sso.common.model.Course

Reason: Spring Data JPA **only scans and registers entities (@Entity)** that are located in the **current application's @EntityScan base packages**

Solution

Explicit **@EntityScan** in the consuming applicatio

```
@EntityScan(basePackages = {
        "com.dxu.sso.common.model",       // 👈 include shared Course entity
        "com.dxu.sso.course.query.model"  // if you have your own entities
})
```

3.2.3 Sopped Loading Configurations from course-config-server

> **Reason**: Did not delete **application.properties** of common-lib which **interferes with the actual microservice's identity** during startup. This confuses the config client, which tries to fetch configuration for the wrong service name.

Solution

Delete **application.properties** for common-lib, or remove property **spring.application.name** in it

3.2.4 Aspect is not triggered

The following aspect is not triggered:

```
@RequireUserProfile
@GetMapping
public ResponseEntity<List<Course>> findCourses(@AuthenticationPrincipal Jwt jwt) { ... }
```

> **Reason**: The method is in a @RestController, not a @Service
>
> Spring AOP by default only proxies **Spring beans**, and only if:
>
> - The bean is **injected through Spring**
> - The call is made **through the proxy** (i.e. external call, not self-call)
> - The aspect is set up correctly

Solution

Enable **proxyTargetClass** mode

```java
@SpringBootApplication
@EnableAspectJAutoProxy(proxyTargetClass = true)
@EntityScan(basePackages = {
        "com.dxu.sso.common.model",       // 👈 include shared Course entity
        "com.dxu.sso.course.query.model"  // if you have your own entities
})
@ComponentScan(basePackages = {
        "com.dxu.sso.course.query",
        "com.dxu.sso.common"           // shared library "common-lib"
})
public class CourseQuerySvcApplication {

    public static void main(String[] args) {
        SpringApplication.run(CourseQuerySvcApplication.class, args);
    }

}
```