
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	6
1. OKTA Setup	7
1. Registration	7
2. Create Users	7
3. Add Claims	7
Steps to Add email as a Custom Claim in Access Token	7
4. Create UserManagementApp Application	7
Step 1: Create 2 OIDC Web Applications in OKTA	7
Step 2: Get the Credentials	8
Step 3: Enable Refresh Token & PKCE	8
5. Create application CourseManagementApp	8
6. Add Custom Claim to Access Token	8
2. Create Gateway and Eureka Server	10
1. Overview	10
2. Design Considerations	10
2.1 Use Single Gateway	10
2.1.2 How to Support Multiple Okta Applications	11
2.2 Use Single Eureka Server	11
2.2.1 Drawbacks of Using Multiple Eureka Servers	11
2.2.2 Benefits of Using a Single Eureka Server	12
3. Implement User Management Application	13

1. Backend Implementation	13
1.1 Overview	13
1.1.1 Authentication & Authorization	13
1.1.2 Components	13
1.2 user-config-server	13
1.3 user-eureka-server	14
1.4 api-gateway	15
1.5 user-profile-svc	18
MySQL:	19
1.6 user-admin-svc	19
1.7 Authentication Test	19
Issue 1:	19
Issue 2	20
Issue 3	21
Issue 4:	22
 4. Implement Course Management Application	 25
1. Backend Implementation	25
1.1 Overview	25
1.1.1 Authentication & Authorization	25
1.1.2 Components	25
1.2 course-config-server	25
1.3 course-eureka-server	26
1.4 course-gateway	27
1.5 course-query-svc	27
MySQL:	29
.....	29
1.6 course-management-svc	29
1.7 course-application-svc	29
 5. Create shared library common-lib	 30
1. Overview	30

2. Implementation	30
2.1 WebClientConfig	32
2.2 ProfileWebClient	32
2.3 Models	33
2.4 DTOs	33
2.5 Security	34
2.5.1 Anotation	34
2.6 Exception Handling	34
2.7 Troubleshooting	36
2.7.1 Unable to find main class on "mvn clean install"	36
.....	37
3. Use common-lib in Microservices	37
3.1 How to Use common-lib in Your Microservices	37
3.2 Troubleshooting During Implementation	37
3.2.1 Could not autowire. No beans of 'ProfileWebClient'	37
3.2.2 IllegalArgumentException - Not a managed type: class com.dxu.sso.common.model.Course	37
3.2.3 Sopped Loading Configurations from course-config-server	38
3.2.4 Aspect is not triggered	38
3.2.5 JPA N+1 query problem	39
4. Best Practices for Shared Libraries like common-lib	42
.....	42
6. Implement Front End	43
6.1 Implement Single Frontend Application	44
1. Create Angular Application course-app	44
1.1 Create Angular 19 App and Install Dependencies	44
1.2 Initialize Components	45
1.3 Implement Others	46
6.2 Implement Separate 2 Front End Applications	59
1. Overview	59

2. Implementation	59
2.1 Front End	59
Step 1: Create a Workspace with Multiple Projects	59
Step 2: Create user-app and shared-lib	60
Step 3: Move Shared Code to shared-lib	60
Step 4: Move User-Related Features to user-app	61
Step 5: Move Course-Related Features to course-app	62
Step 6: Import shared-lib in Both Apps	62
Step 7: Import shared-lib in Both Apps	63
Step 8: External Configuration Injection	63
.....	64
Step 9: Ensure Run Auth Check Before Load Page when Switch to Another App	64
Step 10: Misc Changes	67
2.2 Backend	68
2.2.1 Update api-gateway SecurityConfig to support multiple clients	68
2.2.2 Create a Custom Controller to Initiate Login	70
 7. Architecture and Design	 72
1. Architecture	72
1.1 Architecture Options	72
1.1.1 Should we use PKCE?	72
1.1.2 Communication between FE and BE	72
1.1.3 Inter-Service (Microservices) Communication	72
2. Design	73
2.1 Login Sequence Diagram	73
2.1.1 Actors Involved:	73
2.1.2 Full Authentication Sequence (Authorization Code Flow with PKCE)	74
2.2 Design Approach for Separate Schemas	76
2.2.1 Tables	76
♦ course-enrollment table structure	77

OAuth2 - SSO - Angular - Spring Boot

-  [Attachments](#)
-  [1. OKTA Setup](#)
-  [2. Create Gateway and Eureka Server](#)
-  [3. Implement User Management Application](#)
-  [4. Implement Course Management Application](#)
-  [5. Create shared library common-lib](#)
-  [6. Implement Front End](#)
-  [7. Architecture and Design](#)

1. OKTA Setup

1. Registration

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

daniel.xu4@ontario.ca / Cxxxxxx123!

2. Create Users

student1sso@mailinator.com / Cxxxxxx123!

student2sso@mailinator.com / Cxxxxxx123!

teacher1sso@mailinator.com / Cxxxxxx123!

teacher2sso@mailinator.com / Cxxxxxx123!

admin1sso@mailinator.com / Cxxxxxx123!

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/user-app> (on gateway) (**Changed "okta" to "user-app" because separate gateways does not work and I have to use single gateway. Use "user-app" and "course-app" as registration id for the 2 applications**)
 - **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

6. Add Custom Claim to 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"**

Fill in the following:

Field	Value
Name	firstName
Include in token type	<input checked="" type="checkbox"/> Access Token
Value type	Expression
Value	appUser.given_name
Include in	Scopes: openid email profile

Click **Create**.

To find the field used for the above "Value",

1. **Go to OKTA Admin Console**
2. In the left menu, click **Directory** → **Profile Editor**
3. In the list of **Applications**, find your application: e.g., UserManagementApp

Create the following custom claims:

sub	appuser.userName
email	appuser.email
firstName	appuser.given_name
lastName	appuser.family_name

2. Create Gateway and Eureka Server

1. Overview

- **Single Eureka Server:** All microservices register with one service registry.
- **Single Gateway** (e.g. api-gateway or user-gateway): Handles all routing and security.

2. Design Considerations

2.1 Use Single Gateway

Issue:

1. Originally my Angular frontend uses a proxy config to forward:

- `/api/profile/*` to `http://localhost:9001` (user-gateway)
- `/api/courses/*` to `http://localhost:9011` (course-gateway)

2. **user-gateway** is likely managing the session (holding cookies like `SESSION`) and properly configured for Okta login.

3. **course-gateway** is not sharing the session, so when `/api/courses` is called, it has:

- No access token or session info
- Spring Security thinks you're not logged in → returns 302 redirect to login (`/oauth2/authorization/okta`)

4. Once the browser follows that 302 redirect, it may overwrite cookies, invalidate session, or otherwise interfere with your current login context — causing `/api/profile/me` to stop working too.

Solution:

1. Route all API calls through a **single gateway**, such as **api-gateway**, which ensures:

- All routes go through a **single OAuth2 login session**
- **TokenRelay** forwards the same token to both backend services
- **Session consistency** is maintained

2. Ensure **withCredentials: true** is used in Angular. This ensures **cookies** like **SESSION** are **included** in the request.

```
this.http.get('/api/profile/me', { withCredentials: true })
```

3. Make sure CORS is enabled in all backend services

Each service (course-query-svc, user-profile-svc, etc.) must allow:

- Origin from your Angular dev server (`http://localhost:4200`)
- Allow credentials
- Allow headers like Authorization

4. We are using **OAuth2 Resource Server** on backend, Backends like **course-query-svc** must accept **JWT** via:

```

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

```

2.1.2 How to Support Multiple Okta Applications

Spring Security can distinguish between multiple clients by using **multi-client configuration**:

```





spring:
  security:
    oauth2:
      client:
        registration:
          user-app:
            provider: okta
            client-id: 0oaoogy915xbkRCKT5d7
            client-secret: Ew-4IiHuFr6cgY1w3JhJ5PFWzvB-yZGx6xdD_y3rxhSwioOG-U_NGsCViCv1Axfe
            scope: openid, profile, email
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code /user-app"
          course-app:
            provider: okta
            client-id: 0oaoor5c1w15NU1ef5d7
            client-secret: AC-_PThmy1t69J7Q1Je2g4HNRXq1mPqAUGDspKsy6dTFGk7RVV00a5jKiwZVuJ3z
            scope: openid, profile, email
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/course-app"
        provider:
          okta:
            issuer-uri: https://dev-50623690.okta.com/oauth2/default

```

And in your **gateway security config**, you can customize the OAuth2 login:

2.2 Use Single Eureka Server

2.2.1 Drawbacks of Using Multiple Eureka Servers

Concern	Issue
 Complexity	You need multiple gateways or static routing
 Discovery Isolation	Each Eureka server only knows part of the ecosystem
 Inconsistent Security	Session/cookie/token state may conflict
 Troubleshooting	Harder to trace request flow across domains

2.2.2 Benefits of Using a Single Eureka Server

1. Centralized Service Discovery

- All **microservices** (user-related, course-related, etc.) **register to the same registry**.
- The gateway has a complete view and can route using `lb://service-name`.

2. Clean and Simple Gateway Config

- You don't need to hardcode static URIs or worry about manual routing.
- Just use `lb://course-query-svc` or `lb://user-profile-svc`.

3. Better Load Balancing and Failover

- Spring Cloud LoadBalancer works best with Eureka.
- Failover, health checks, retries, and service instance awareness are automatic.

4. Better Observability

- You can view all microservices and their status in one Eureka dashboard.

5. Simpler Configuration Management

- You can use a unified Spring Cloud Config Server or `application.yml` setup across services.

6. Less Error-Prone Security Handling

- Session, token relay, and login logic (especially with Okta/OAuth2) are more consistent.

3. Implement User Management Application

1. Backend Implementation

1.1 Overview

1.1.1 Authentication & Authorization

- **OAuth2 login via OKTA** is handled **only by api-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
api-gateway	Central entry point (handles login & token relay), SINGLE Gateway	9001
user-profile-svc	Saves profile of the logged-in OKTA user	9002
user-admin-svc	Allows admin to manage user accounts	9003
user-config-server	Centralized configuration	9004
eureka-server	SINGLE 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

```
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

- eureka-server.yml
- api-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

```
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 api-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

```

server:
  port: 9001

spring:
  application:
    name: api-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: 0oaoqy915xbkRCKT5d7
          client-secret: Ew-4IiHuFr6cgY1w3JhJ5PFWzvB-yZGx6xdD_y3rxhSwio0G-U_NGsCViCv1Axfe
          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

angular:
  redirect_url: http://localhost:4200

```

SecurityConfig.java

3. Implement User Management Application

```
@Configuration
@EnableWebFluxSecurity
public class SecurityConfig {

    @Value( "${angular.redirect-url}")
    private String angularRedirectUrl;    // This must be one of the

    @Bean
    public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity http,
                                                            ReactiveClientRegistrationRepository
clientRegistrationRepository) {
        return http
            .csrf(csrf -> csrf.disable())
            .authorizeExchange(authorize -> authorize
                .pathMatchers("/actuator/**", "/public/**").permitAll()
                .anyExchange().authenticated())
            .oauth2Login(login -> login
                .authenticationSuccessHandler(redirectToAngular()))
            .logout(logout -> logout
                .logoutSuccessHandler(oidcLogoutSuccessHandler(clientRegistrationRepository)))
            .build(); // ✅ No more .oauth2Client()
    }

    private ServerLogoutSuccessHandler oidcLogoutSuccessHandler(
        ReactiveClientRegistrationRepository clientRegistrationRepository) {

        OidcClientInitiatedServerLogoutSuccessHandler handler =
            new OidcClientInitiatedServerLogoutSuccessHandler(clientRegistrationRepository);

        // Redirect back to Angular after logging out of Okta
        handler.setPostLogoutRedirectUri(String.valueOf(URI.create(angularRedirectUrl)));
        return handler;
    }

    private ServerAuthenticationSuccessHandler redirectToAngular() {
        return new RedirectServerAuthenticationSuccessHandler(angularRedirectUrl);
    }
}
```

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

i 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

```
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 'passwd';
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:

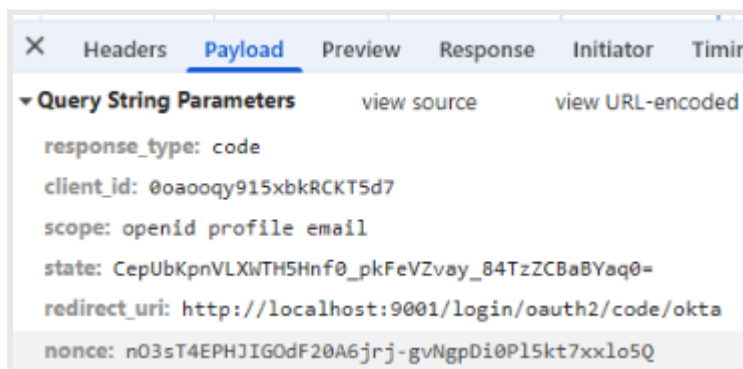
Expected: Redirect to OKTA login page

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=0oaoqy915xbkRCKT5d7&scope=openid%20profile%20email&state=CepUbKpnVLXWTH5Hnf0_pkFeVZvay_84TzZCBaBYaq0%3D&redirect_uri=http://localhost:9001/login/oauth2/code/okta&nonce=nO3sT4EPHJIGOdF20A6jrj-gvNgpDi0PI5kt7xxlo5Q

The above URL of the 302 location



400

Bad Request

Your request resulted in an error. Policy evaluation failed for this request, please check the policy configurations.

Reason: caused by **missing or misconfigured access policy rules** on OKTA's **Authorization Server**, specifically for your OIDC Web App

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 api-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 — api-gateway / client ID: 0oaoqy915xbkRCKT5d7)
3. Click **Create Rule**

Issue 2

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

Reason:

- api-gateway cannot resolve the host name via DNS
- it relies on Eureka for routing (using `lb://user-profile-svc`), and the system fails because the actual network host is unknown or unreachable

Solution: Force Eureka to Register localhost or IP Instead

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

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

This tells Eureka to:

- Register using your machine's **IP address or localhost**
- Avoid relying on system hostnames (like dxpsdesktop.mshome.net)

Issue 3

Error:

In **URL:** https://dev-50623690.okta.com/oauth2/default/v1/authorize?response_type=code&client_id=0oaooy915xbkRCKT5d7&scope=openid%20profile%20email&state=L_yJq5kOzBNHCfBKGrKy6zNLM7KFXO9_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/0oaooy915xbkRCKT5d7#tab-general

Reason:

From OKTA's point of view:

- Your app sent this in the authorization request: **redirect_uri=http://172.21.48.1:9002/login/oauth2/code/okta**

But in your OKTA app settings, the allowed **Sign-in redirect URI** is:

- <http://localhost:9001/login/oauth2/code/okta>

➔ These **do not match**, and OKTA rejects the request.

Solution: Only let Gateway Handles Login

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

```
spring:
  security:
    oauth2:
      client:
        registration:
          okta:
            client-id: 0oaooy915xbkRCKT5d7
            client-secret: Ew-4IiHuFr6cgY1w3JhJ5PFwzvB-yZGx6xdD_y3rxhSWio0G-U_NgSCViCv1Axfe
            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 in `user-profile-svc` / `user-admin-svc` 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.boot</groupId>
  <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

Also, Spring Security needs a **JwtDecoder bean**. **Below** let Spring Boot auto-configure JwtDecoder using the OKTA issuer URI

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

Recap What's Happening

Now that you've added in **SecurityConfig** in **user-profile-svc**:

```
.oauth2ResourceServer(oauth2 -> oauth2.jwt(Customizer.withDefaults()))
```

This tells Spring Security to validate incoming requests using JWT tokens, and it auto-wires the logic behind the scenes.

Spring Security expects to **decode JWT tokens** from incoming requests (relayed from your gateway).

To do this, it needs to know:

- **Where** to fetch the public keys (JWKS) from
- **How** to decode and validate the access token

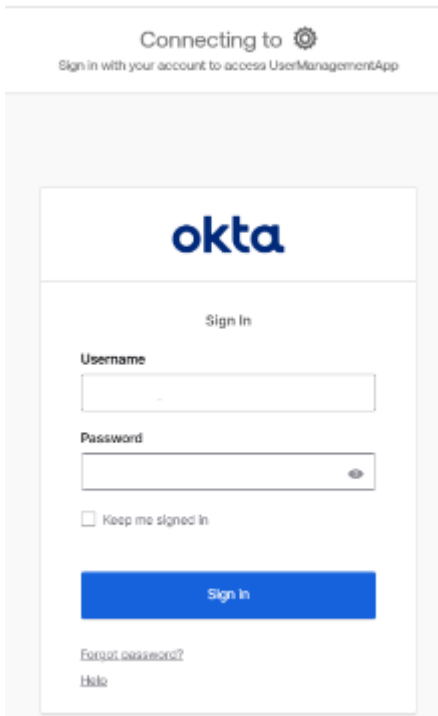
By providing the issuer-uri, Spring Boot will:

- Auto-configure a JwtDecoder bean
- Download your OKTA public keys from:


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

3. Implement User Management Application

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



The image shows the Okta login page. At the top, it says "Connecting to" followed by a gear icon and "Sign in with your account to access UserManagementApp". Below this is the Okta logo. The main section is titled "Sign in" and contains a "Username" field, a "Password" field with a toggle icon, a "Keep me signed in" checkbox, and a blue "Sign in" button. At the bottom, there are links for "Forgot password?" and "Help".

Connecting to 
Sign in with your account to access UserManagementApp

okta

Sign in

Username

Password

☐ Keep me signed in

[Sign in](#)

[Forgot password?](#)
[Help](#)

4. Implement Course Management Application

1. Backend Implementation

1.1 Overview

1.1.1 Authentication & Authorization

- **OAuth2 login via OKTA** is handled **only by api-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-query-svc	9012	Read-only API for all roles
course-management-svc	9013	Admin/teacher APIs for course updates
course-application-svc	9014	Student course application + approval
config-server (shared)	9015	Centralized config
MySQL (shared)	—	Stores course, enrollment, and application data

1.2 course-config-server

Dependencies:

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

application.yml

```

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

```

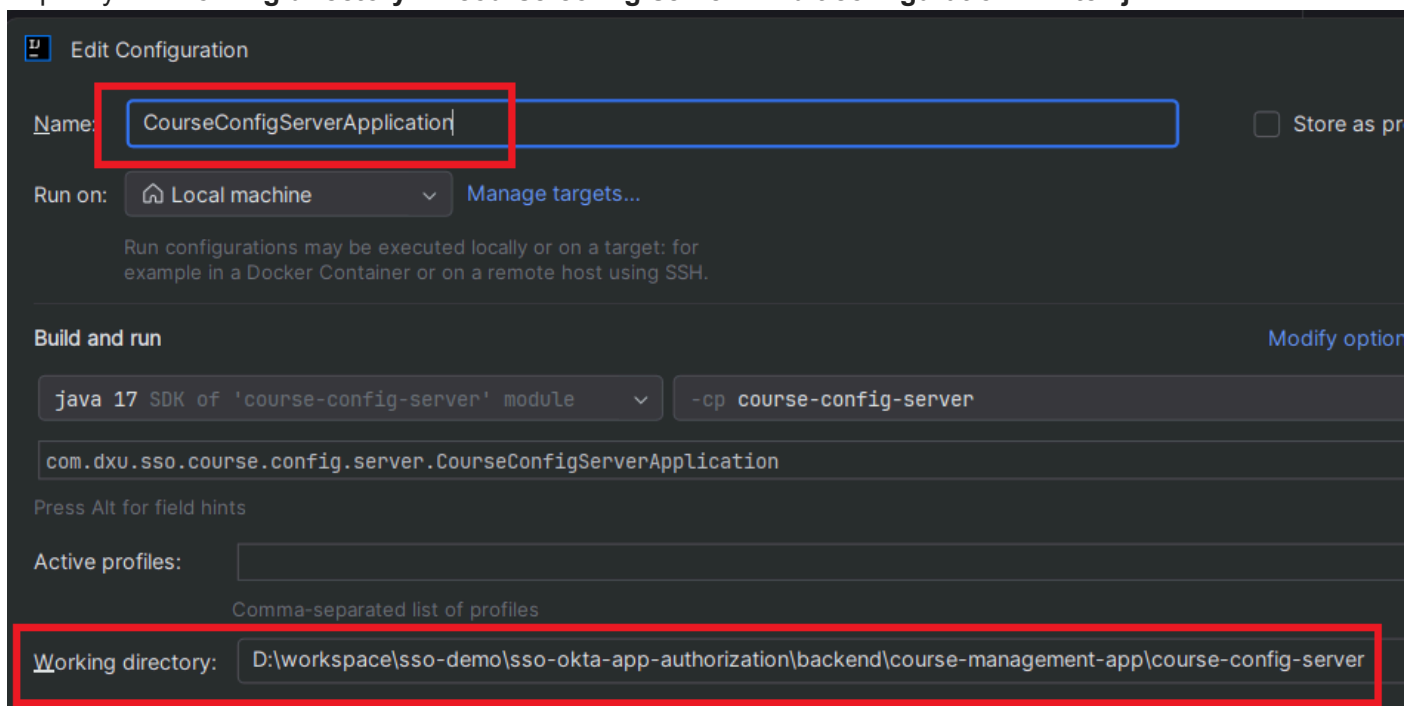
@EnableConfigServer

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** in IntelliJ IDEA



1.3 ~~course-eureka-server~~

⚠ Removed since we use **SINGLE** eureka server to prevent complexity

~~1.4 course-gateway~~

Removed since we use **SINGLE** gateway to prevent complexity

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

```
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

```

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: password

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

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

```

@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 'password';  
GRANT ALL PRIVILEGES ON ssocourse.* TO 'ssocourse'@'localhost';  
FLUSH PRIVILEGES;
```

1.6 course-management-svc

Mostly same as course-query-svc

1.7 course-application-svc

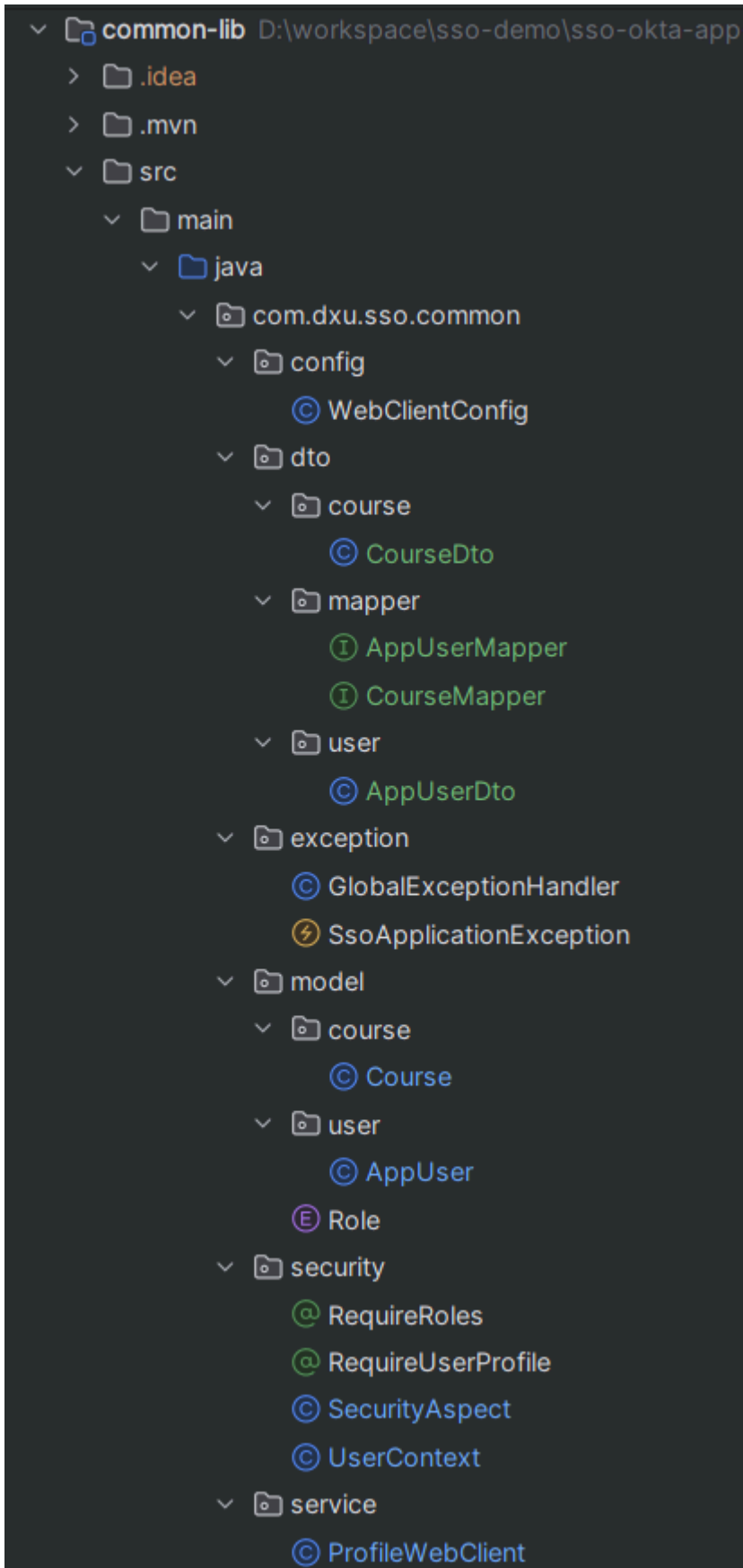
Mostly same as course-query-svc and course-management-svc

5. 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



- **WebClientConfig**: Create WebClient.builder that is shared by applications to do the **inter-application communication**
- **GlobalExceptionHandler**: The global exception handler
- **dto**: Shared DTOs, mapper, etc
- **model**: Shared entity

- **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.

Note: Need to set Authorization header with bearer token

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

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

    String authHeader = getAuthHeader();
    if (authHeader == null || !authHeader.startsWith("Bearer ")) return null;

    AppUserDto user = webClientBuilder.build()
        .get()
        .uri(userProfileUrl)
        .header(HttpHeaders.AUTHORIZATION, authHeader)
        .retrieve()
        .onStatus(HttpStatus::is4xxClientError, response -> Mono.empty())
        .bodyToMono(AppUserDto.class)
        .block();

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

private static String getAuthHeader() {
    ServletRequestAttributes attrs = (ServletRequestAttributes)
RequestContextHolder.getRequestAttributes();
    if (attrs == null) return null;

    return attrs.getRequest().getHeader(HttpHeaders.AUTHORIZATION);
}
```


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

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

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

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

        ... ..
        AppUserDto user = ... ..;

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

2.3 Models

Shared JPA entities.

⚠ Do not share entities across services unless they use the same schema. Misusing might cause the same entity creates DB table in multiple databases (schemas)

In our case, we have the following entities created in common-lib:

- com.dxu.sso.common.model.**user**.AppUser
- com.dxu.sso.common.model.**course**.Course
- com.dxu.sso.common.model.**course**.CourseEnrollment

We create them in different packages, then we can scan them separately in necessary microservices

2.4 DTOs

Shared DTOs, mappers, etc.

2.5 Security

2.5.1 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`

```
@Aspect
@Component
@RequiredArgsConstructor
public class SecurityAspect {

    private final ProfileWebClient profileWebClient;

    @Around("@annotation(requireUserProfile)")
    public Object checkUserProfile(ProceedingJoinPoint joinPoint, RequireUserProfile
requireUserProfile)
        throws Throwable {
        AppUserDto user = profileWebClient.getUserProfile();
        if (user == null) {
            throw new SsoApplicationException(HttpStatus.FORBIDDEN.value(), "User profile not
found");
        }
        return joinPoint.proceed();
    }

    @Around("@annotation(requireRoles)")
    public Object checkUserRole(ProceedingJoinPoint joinPoint, RequireRoles requireRoles) throws
Throwable {
        AppUserDto user = profileWebClient.getUserProfile();
        if (user == null) {
            throw new SsoApplicationException(HttpStatus.FORBIDDEN.value(), "User profile not
found");
        }

        List<String> allowedRoles = Arrays.asList(requireRoles.value());
        if (!allowedRoles.contains(user.getRole())) {
            throw new SsoApplicationException(HttpStatus.FORBIDDEN.value(), "User role not
permitted");
        }

        return joinPoint.proceed();
    }
}
```

2.6 Exception Handling

- `SsoApplicationException`: The global exception handler shared by applications

Note: SsoApplicationException **MUST** extends RuntimeException.

Spring AOP uses **dynamic proxies** by default (JDK proxies or CGLIB), and **unchecked exceptions (like RuntimeException) are passed through**, but **checked exceptions (or any custom exception that isn't declared) get wrapped** in UndeclaredThrowableException. This tells Spring AOP and Java proxying:

“This is an **unchecked exception**, don't wrap it — just let it bubble.”

```
@Getter
@Setter
public class SsoApplicationException extends RuntimeException {

    private final int statusCode;

    public SsoApplicationException(int statusCode, String message) {
        super(message);
        this.statusCode = statusCode;
    }
}
```

```

@Slf4j
@AllArgsConstructor
@ControllerAdvice
public class GlobalExceptionHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler({ SsoApplicationException.class })
    public ResponseEntity<?> handleApplicationException(SsoApplicationException ex) {
        log.error("GlobalExceptionHandler: ", ex);

        return ResponseEntity
            .status(ex.getStatusCode())
            .body(new ErrorResponse(ex.getStatusCode(), ex.getMessage()));
    }

    @ExceptionHandler({ Exception.class })
    @ResponseBody
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    public ResponseEntity<?> handleGenericException(Exception ex, WebRequest req) {
        log.error("Exception: ", ex);

        return ResponseEntity
            .status(HttpStatus.INTERNAL_SERVER_ERROR)
            .body(new ErrorResponse(HttpStatus.INTERNAL_SERVER_ERROR.value(), ex.getMessage()));
    }

    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException
ex,

                                                                    HttpHeaders headers,
                                                                    HttpStatusCode status,
                                                                    WebRequest request) {

        Map<String, String> errors = new HashMap<>();
        ex.getBindingResult().getAllErrors().forEach((error) -> {
            String fieldName = ((FieldError) error).getField();
            String errorMessage = error.getDefaultMessage();
            errors.put(fieldName, errorMessage);
        });

        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(errors);
    }

    record ErrorResponse(int statusCode, String message) {}
}

```

2.7 Troubleshooting

2.7.1 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**.
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

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

- Enable AOP in each microservice

```
@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 application

```
@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

```

@SpringBootApplication
@EnableAspectJAutoProxy(proxyTargetClass = true)
@EntityScan(basePackages = {
    "com.dxu.sso.common.model.course", // 👉 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);
    }

}

```

3.2.5 JPA N+1 query problem

❗ For the following entities definitions, there is JPA N+1 query problem because, by default, JPA lazily loads the collection, resulting in:

- 1 query to fetch all courses
- 1 separate query per course to fetch enrolledStudentIds from course_enrollment

```

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Table(name = "course", schema = "ssocourse", uniqueConstraints = {
    @UniqueConstraint(columnNames = "name")
})
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String description;
    private LocalDate startDate;
    private LocalDate endDate;

    @JoinColumn(name = "teacher_id")
    private Long teacherId;    // FK reference to AppUser.id (role=TEACHER)

    @ElementCollection
    @CollectionTable(name = "course_enrollment", schema = "ssocourse", joinColumns = @JoinColumn(name =
"course_id"))
    @Column(name = "student_id")
    private List<Long> enrolledStudentIds = new ArrayList<>();
}

@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Entity
@Table(name = "course_enrollment", schema = "ssocourse")
public class CourseEnrollment {

    @EmbeddedId
    private CourseEnrollmentId id;

    @Column(name = "created_at")
    private LocalDateTime createdAt = LocalDateTime.now();
}

@Embeddable
public class CourseEnrollmentId implements Serializable {

    @Column(name = "course_id")
    private Long courseId;

    @Column(name = "student_id")
    private Long studentId;
}

```

Solution:

Replace **@ElementCollection** with an explicit **@OneToMany** relationship in **Course.java**


```
@OneToMany(mappedBy = "course", cascade = CascadeType.ALL, orphanRemoval = true)
private List<CourseEnrollment> enrollments = new ArrayList<>();
```

Updated CourseEnrollment.java

```
@Entity
@Table(name = "course_enrollment", schema = "ssocourse")
public class CourseEnrollment {

    @EmbeddedId
    private CourseEnrollmentId id;

    @ManyToOne(fetch = FetchType.LAZY)
    @MapsId("courseId") // Maps to embedded ID field
    @JoinColumn(name = "course_id")
    private Course course;

    @Column(name = "created_at")
    private LocalDateTime createdAt = LocalDateTime.now();

    public Long getStudentId() {
        return id != null ? id.getStudentId() : null;
    }
}
```

Add findAllWithEnrollments query in the CourseRepository:

```
@Repository
public interface CourseRepository extends JpaRepository<Course, Long> {

    @Query("SELECT c FROM Course c LEFT JOIN FETCH c.enrollments")
    List<Course> findAllWithEnrollments();
}
```

Revise CourseService.updateCourse()

```

public CourseDetailsDto updateCourse(Long id, CourseSaveRequest request) {
    Course course = courseRepository.findById(id)
        .orElseThrow(() -> new SsoApplicationException(HttpStatus.BAD_REQUEST.value(), "Course not
found"));

    course.setName(request.getName());
    course.setDescription(request.getDescription());
    course.setStartDate(request.getStartDate());
    course.setEndDate(request.getEndDate());
    course.setTeacherId(request.getTeacherId());

    // ✨ Rebuild enrollment list
    List<CourseEnrollment> newEnrollments = request.getEnrolledStudentIds() != null
        ? request.getEnrolledStudentIds().stream()
            .map(studentId -> CourseEnrollment.builder()
                .id(new CourseEnrollmentId(course.getId(), studentId))
                .course(course)
                .build())
            .toList()
        : new ArrayList<>();

    // ⚠️ Clear and replace enrollments
    course.getEnrollments().clear();
    course.getEnrollments().addAll(newEnrollments);

    Course updated = courseRepository.save(course);
    return getCourseDetails(updated);
}

```

4. Best Practices for Shared Libraries like common-lib

- ❌ Do not include application.properties or application.yml in common-lib
 - Even empty files can override parent config
- ✅ Move all configuration to the microservices or external config repo
- ✅ Use @Value, @ConfigurationProperties, or @EnableConfigurationProperties in common-lib to read properties — **but define the values externally** (in the consuming service)
- ❌ Do not share entities across services unless they use the same schema.

6. Implement Front End

6.1 Implement Single Frontend Application

1. Create Angular Application course-app

1.1 Create Angular 19 App and Install Dependencies

```
npm install @angular/cli@19.0.1

ng new course-app --standalone --routing --style=scss

npm install @angular/material @angular/cdk bootstrap
```

angular.json

```
"styles": [
  "node_modules/bootstrap/dist/css/bootstrap.min.css",
  "node_modules/@angular/material/prebuilt-themes/indigo-pink.css",
  "src/styles.scss"
]
```

main.ts

```
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, appConfig)
  .catch((err) => console.error(err));
```

app.config.ts

```

import { ApplicationConfig, importProvidersFrom, provideZoneChangeDetection } from '@angular/core';
import { provideRouter } from '@angular/router';
import { HTTP_INTERCEPTORS, provideHttpClient, withInterceptorsFromDi } from '@angular/common/http';
import { BrowserAnimationsModule, provideAnimations } from '@angular/platform-browser/animations';
import { MatSnackBarModule } from '@angular/material/snack-bar';

import { routes } from './app.routes';
import { AppInterceptor } from './shared/core/app.interceptor';
import { DateFormatProvider } from './shared/core/date-format.provider';
import { MatNativeDateModule } from '@angular/material/core';

export const appConfig: ApplicationConfig = {
  providers: [
    provideZoneChangeDetection({ eventCoalescing: true }),
    provideHttpClient(withInterceptorsFromDi()), // ✅ This fixes HttpClient injection
    provideRouter(routes),
    {
      provide: HTTP_INTERCEPTORS,
      useClass: AppInterceptor,
      multi: true
    },
    importProvidersFrom(
      BrowserAnimationsModule,
      MatSnackBarModule
    ),
    provideRouter(routes),
    provideAnimations(),
    importProvidersFrom(MatNativeDateModule), // For Date format
    DateFormatProvider // For Date format
  ],
};

```

1.2 Initialize Components

```

ng generate component layout/header --standalone
ng generate component dashboard --standalone
ng generate component user/user-profile --standalone
ng generate component user/user-list --standalone
ng generate component course/course-list --standalone
ng generate component course/course-details --standalone

```

app.routes.ts

```
export const routes: Routes = [
  { path: '', component: DashboardComponent },
  // { path: 'profile', loadComponent: () => import('./user/user-profile/user-profile.component').then(m
=> m.UserProfileComponent), canActivate: [AuthGuard] },
  { path: 'profile', component: UserProfileComponent, canActivate: [ AuthGuard ] },
  { path: 'admin/users', component: UserListComponent, canActivate: [ AuthGuard, RoleGurad ], data: {
roles: ['ADMIN']} },
  { path: 'admin/users/:id', component: UserProfileComponent, canActivate: [ AuthGuard, RoleGurad ], data:
{ roles: ['ADMIN', "TEACHER"]} },
  { path: 'courses', component: CourseListComponent, canActivate: [ AuthGuard ] },
  { path: 'courses/manage', component: CourseDetailsComponent, canActivate: [ AuthGuard, RoleGurad ],
data: { roles: ['ADMIN']} }
];
```

app.component

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterModule, HeaderComponent],
  template: `
    <app-header></app-header>
    <div class="container mt-4">
      <router-outlet></router-outlet>
    </div>
  `
})
export class AppComponent {}
```

1.3 Implement Others

header.component

```
export const LOGIN_URL = `${environment.userBackendHost}/oauth2/authorization/user-app`;
```

```

export class HeaderComponent {

    logoutUrl: string = `${environment.userBackendHost}/logout`

    constructor(private authService: AuthService) {}

    isAuthenticated() {
        return !!this.authService.profile;
    }

    isAdmin() {
        return this.authService.profile?.role === Role.ADMIN;
    }

    login(): void {
        window.location.href = LOGIN_URL;
    }

    logout(): void {
        const form = document.getElementById('logoutForm') as HTMLFormElement;
        form?.submit(); // ✅ Spring handles the logout, redirect, and session invalidation
    }
}

```

```

<nav class="navbar navbar-expand-lg navbar-dark bg-primary">
  <div class="container-fluid">
    <a class="navbar-brand" routerLink="/">Course Management</a>

    <div class="d-flex ms-auto" *ngIf="isAuthenticated(); else guest">
      <a class="btn btn-light btn-sm me-2" routerLink="/profile">My Profile</a>

      <!-- Role-based options (ADMIN / TEACHER / STUDENT) -->
      <a class="btn btn-warning btn-sm me-2" *ngIf="isAdmin()" routerLink="/admin/users">Admin</a>
      <a class="btn btn-light btn-sm me-2" routerLink="/courses">Courses</a>

      <!-- ✅ Logout button -->
      <button class="btn btn-outline-light btn-sm" (click)="logout()">Logout</button>

      <!-- ✅ Hidden Logout form -->
      <form id="logoutForm" method="POST" action="{{logoutUrl}}" style="display: none;"></form>
    </div>

    <ng-template #guest>
      <button class="btn btn-outline-light btn-sm" (click)="login()">Login</button>
    </ng-template>
  </div>
</nav>

```

profile.service.ts

```
getProfile(): Observable<any> {  
    return this.http.get<any>(URL_PROFILE, { withCredentials: true });  
}  
  
createProfile(): Observable<any> {  
    return this.http.post<any>(URL_PROFILE, null, { withCredentials: true });  
}  
  
updateProfile(user: UserForm): Observable<any> {  
    return this.http.put<UserForm>(URL_PROFILE, user, { withCredentials: true });  
}
```

auth.service.ts


```

export class AuthService {

  private authenticated = new BehaviorSubject<boolean>(false);
  public isAuthenticated$ = this.authenticated.asObservable();

  private _profile: any = null;

  public get profile() {
    return this._profile;
  }

  constructor(private profileService: ProfileService, private router: Router) {
  }

  checkAuthentication(): void {
    this.profileService.getProfile().subscribe({
      next: (user) => {
        this._profile = user;
        this.authenticated.next(true);
      },
      error: (error: HttpResponse) => {
        if (error.status === 404) {
          // Profile not found → try creating it
          this.profileService.createProfile().subscribe({
            next: (createdUser) => {
              this._profile = createdUser;
              this.authenticated.next(true);
              this.router.navigate(['/']); // redirect to home after profile creation
            },
            error: (err) => {
              this._profile = null;
              this.authenticated.next(false);
              console.error('❌ Failed to create profile', err);
              this.router.navigate(['/']); // still redirect to home
            }
          });
        } else {
          this._profile = null;
          this.authenticated.next(false);
          console.error('❌ Failed to load profile', error);
        }
      }
    });
  }
}

```

auth.guard.ts

```

export const AuthGuard: CanActivateFn = () => {
  const authService = inject(AuthService);

  return authService.isAuthenticated$.pipe(
    map(isAuth => {
      if (!isAuth) {
        window.location.href = LOGIN_URL;
        return false;
      }
      return true;
    })
  );
};

```

role.guard.ts

```

export const RoleGuard: CanActivateFn = (route, state) => {
  const authService = inject(AuthService);
  const router = inject(Router);
  const snackBar = inject(MatSnackBar);

  const allowedRoles: string[] = route.data['roles'] || [];
  const userRole = authService.profile?.role;

  if (!userRole || !allowedRoles.includes(userRole)) {
    snackBar.open('⛔ You are not authorized to access this page.', 'Dismiss', {
      duration: 5000,
      verticalPosition: 'top',
      panelClass: 'mat-mdc-snack-bar-warn'
    });

    router.navigate(['/']);
    return false;
  }

  return true;
}

```

app.interceptor.ts

```

@Inject()
export class AppInterceptor implements HttpInterceptor {
  private snackBar = inject(MatSnackBar);
  private router = inject(Router);

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    return next.handle(req).pipe(
      catchError((error: HttpResponse) => {
        if (error.status === 401) {
          // ✓ This for logged in user who does not have profile
          this.snackBar.open('⚠ You do not have profile on our site.', 'Dismiss', {
            duration: 5000,
            verticalPosition: 'top',
            panelClass: 'mat-mdc-snack-bar-warn'
          });
          // ✓ No profile. Go to dashboard page
          this.router.navigate(['']);
        } else if (error.status === 403) {
          // ✓ Not authorized – show snackbar
          this.snackBar.open('⚠ You are not authorized to access this resource.', 'Dismiss', {
            duration: 5000,
            verticalPosition: 'top',
            panelClass: 'mat-mdc-snack-bar-warn'
          });
          // ✓ No profile. Go to dashboard page
          this.router.navigate(['']);
        }

        return throwError(() => error);
      })
    );
  }
}

```

user-profile.component (Shared by profile and admin)

```

export class UserProfileComponent implements OnInit, OnDestroy {

  private route = inject(ActivatedRoute);
  private router = inject(Router);
  private profileService = inject(ProfileService);
  private adminService = inject(UserAdminService);
  private notification = inject(NotificationService);
  private dateAdapter = inject(DateAdapter<Date>);

  private destroy$ = new Subject<void>();
  private profileData: any;

  userId?: number;
  isEditMode: boolean = false;

  form!: FormGroup;

  ngOnInit(): void {
    this.dateAdapter.setLocale('en-CA'); // ensures yyyy-MM-dd format

    this.userId = Number(this.route.snapshot.paramMap.get('id'));
    this.route.paramMap.subscribe(params => {
      const id = params.get('id');
      if (id) {
        // Fetch user's profile by admin
        this.loadUser(+id);
      } else {
        // Load self profile
        this.loadSelf();
      }
    });
  }

  loadUser(id: number): void {
    this.adminService.getUserById(id)
      .pipe(takeUntil(this.destroy$))
      .subscribe({
        next: user => {
          this.profileData = user;
          this.form = new User(user).toForm();
        },
        error: () => {
          this.notification.error('Failed to load user information');
          this.router.navigate(['/admin/users']);
        }
      });
  }

  loadSelf(): void {
    this.profileService.getProfile()
      .pipe(takeUntil(this.destroy$))
      .subscribe({
        next: profile => {
          this.profileData = profile;
          this.form = new User(profile).toForm();
        },
        error: () => {
          this.notification.error('Failed to load profile');
          this.router.navigate(['/admin/users']);
        }
      });
  }

```

```

}

submit(): void {
  this.form.markAllAsTouched();
  if (this.form.valid) {
    const profile = new User().toModel(this.form);
    const payload = {
      ...profile,
      dateOfBirth: profile.dateOfBirth ? format(profile.dateOfBirth, 'yyyy-MM-dd') : null
    };

    if (this.userId) {
      this.adminService.updateUserByAdmin(this.userId, payload).subscribe({
        next: profile => {
          this.profileData = profile;
          this.toggleEdit();
          this.notification.success('User information updated successfully');
        },
        error: err => this.notification.error('Failed to update profile')
      })
    } else {
      this.profileService.updateProfile(payload).subscribe({
        next: profile => {
          this.profileData = profile;
          this.toggleEdit();
          this.notification.success('Profile updated successfully');
        },
        error: err => this.notification.error('Failed to update profile')
      })
    }
  }
}

toggleEdit() {
  if (this.isEditMode) {
    this.form = new User(this.profileData).toForm();
  }
  this.isEditMode = !this.isEditMode;
}

cancelEdit() {
  this.form.markAsPristine();
  this.isEditMode = false;
  this.form.patchValue(this.profileData); // Restore previous values
}

ngOnDestroy(): void {
  this.destroy$.next();
  this.destroy$.complete();
}

```

```

<div class="container mt-4">
  <h2 class="mb-4">User Profile</h2>

  <div *ngIf="!isEditMode">
    <div class="row mb-3">
      <div class="col-md-12 mt-2"><strong>OKTA ID:</strong> {{ oktaUserId?.value }}</div>
      <div class="col-md-6 mt-2"><strong>First Name:</strong> {{ firstName?.value }}</div>
      <div class="col-md-6 mt-2"><strong>Last Name:</strong> {{ lastName?.value }}</div>
      <div class="col-md-6 mt-2"><strong>Email:</strong> {{ email?.value }}</div>
      <div class="col-md-6 mt-2"><strong>Date of Birth:</strong> {{ dateOfBirth?.value | date:'yyyy-MM-dd'
    }}</div>
    <div class="col-md-6 mt-2"><strong>Gender:</strong>
      <ng-container [ngSwitch]="gender?.value">
        <span *ngSwitchCase="'M'">Male</span>
        <span *ngSwitchCase="'F'">Female</span>
        <span *ngSwitchDefault>Other</span>
      </ng-container>
    </div>
    <div class="col-md-6 mt-2 mb-2"><strong>Role:</strong> {{ role?.value }}</div>
  </div>
  <button mat-raised-button color="primary" (click)="toggleEdit()">Edit</button>
</div>

<form *ngIf="isEditMode" [formGroup]="form" (ngSubmit)="submit()" class="row g-3">
  <div class="col-md-6">
    <mat-form-field appearance="outline" class="w-100">
      <mat-label>First Name</mat-label>
      <input matInput formControlName="firstName" required />
      <mat-error *ngIf="form.get('firstName')?.hasError('required')">First Name is required</mat-error>
    </mat-form-field>
  </div>

  <div class="col-md-6">
    <mat-form-field appearance="outline" class="w-100">
      <mat-label>Last Name</mat-label>
      <input matInput formControlName="lastName" required />
      <mat-error *ngIf="form.get('lastName')?.hasError('required')">Last Name is required</mat-error>
    </mat-form-field>
  </div>

  <div class="col-md-6">
    <mat-form-field appearance="outline" class="w-100">
      <mat-label>Email</mat-label>
      <input matInput formControlName="email" required />
      <mat-error *ngIf="form.get('email')?.hasError('required')">Email is required</mat-error>
    </mat-form-field>
  </div>

  <div class="col-md-6">
    <mat-form-field appearance="outline" class="w-100">
      <mat-label>Date of Birth</mat-label>
      <input matInput [matDatepicker]="picker" formControlName="dateOfBirth">
      <mat-datepicker-toggle matSuffix [for]="picker"></mat-datepicker-toggle>
      <mat-datepicker #picker></mat-datepicker>
    </mat-form-field>
  </div>

  <div class="col-md-6">
    <mat-form-field appearance="outline" class="w-100">

```

```

    <mat-option value="M">Male</mat-option>
    <mat-option value="F">Female</mat-option>
    <mat-option value="OTH">Prefer Not to Say</mat-option>
  </mat-select>
  <mat-error *ngIf="form.get('gender')?.hasError('required')">Gender is required</mat-error>
</mat-form-field>
</div>

<div class="col-md-6">
  <mat-form-field appearance="outline" class="w-100">
    <mat-label>Role</mat-label>
    <mat-select formControlName="role" required>
      <mat-option value="NONE">None</mat-option>
      <mat-option value="STUDENT">Student</mat-option>
      <mat-option value="TEACHER">Teacher</mat-option>
      <mat-option value="ADMIN">Admin</mat-option>
    </mat-select>
    <mat-error *ngIf="form.get('role')?.hasError('required')">Role is required</mat-error>
  </mat-form-field>
</div>

<div class="col-12 d-flex justify-content-start gap-2">
  <button mat-raised-button color="primary" type="submit" [disabled]="form.invalid">Save</button>
  <button mat-raised-button type="button" (click)="cancelEdit()">Cancel</button>
</div>
</form>
</div>

```

Create a service to handling message displaying

```

export class NotificationService {

  constructor(private snackBar: MatSnackBar) {}

  success(message: string): void {
    this.snackBar.open(`✅ ${message}`, 'Dismiss', {
      duration: 3000,
      verticalPosition: 'top',
      panelClass: ['mat-mdc-snack-bar-success']
    });
  }

  error(message: string): void {
    this.snackBar.open(`❌ ${message}`, 'Dismiss', {
      duration: 5000,
      verticalPosition: 'top',
      panelClass: ['mat-mdc-snack-bar-error']
    });
  }

  warning(message: string): void {
    this.snackBar.open(`⚠️ ${message}`, 'Dismiss', {
      duration: 5000,
      verticalPosition: 'top',
      panelClass: ['mat-mdc-snack-bar-warn']
    });
  }

  info(message: string): void {
    this.snackBar.open(`ℹ️ ${message}`, 'Dismiss', {
      duration: 4000,
      verticalPosition: 'top',
      panelClass: ['mat-mdc-snack-bar-info']
    });
  }
}

```

Handling Date Format Issues

1. Submit in Format of yyyy-MM-dd

Currently the dateOfBirth sent from UI does not match the format required by REST API which is yyyy-MM-dd. Fix it as below

- Install **date-fns**

```
npm i date-fns
```

- Convert in submit()


```
const profile = new UserForm().toModel(this.form);
const payload = {
  ...profile,
  dateOfBirth: profile.dateOfBirth ? format(profile.dateOfBirth, 'yyyy-MM-dd') : null
};
```

2. Display Format in the DatePicker

✓ Step 1: Configure Global Date Format

```
import { MAT_DATE_FORMATS } from '@angular/material/core';

export const APP_DATE_FORMATS = {
  parse: {
    dateInput: 'YYYY-MM-DD',
  },
  display: {
    dateInput: 'YYYY-MM-DD',
    monthYearLabel: 'MMM YYYY',
    dateA11yLabel: 'LL',
    monthYearA11yLabel: 'MMMM YYYY',
  }
};

export const DateFormatProvider = {
  provide: MAT_DATE_FORMATS,
  useValue: APP_DATE_FORMATS,
};
```

✓ Step 2: Register the Provider in app.config.ts

```
export const appConfig: ApplicationConfig = {
  providers: [
    ...
    DateFormatProvider
  ],
};
```

3. Wrong Date Displayed in DatePicker

❗ `new Date('yyyy-MM-dd')` results in a date that is **one day earlier**

After fetched `dateOfBirth` from REST API in format `yyyy-MM-dd`, when use `new Date(dateOfBirth)`, it always results in the date that is one day earlier.

Reason:

When you pass a date string like `'2024-10-01'` to `new Date()`, JavaScript interprets it as **midnight UTC** (i.e., `2024-10-01T00:00:00.000Z`). Then it converts that to **local time**, which could be **the previous day** if your time zone is behind UTC.

Solution:

Parse the date string using a **library that treats it as a local date**, such as **date-fns**

user-from.model.ts

```
toForm() {  
  return new FormGroup({  
    ... ..,  
    dateOfBirth: new FormControl(this.toDate(this.dateOfBirth), Validators.required)  
  })  
}  
  
toDate(dateOfBirth: any) {  
  if (!dateOfBirth) {  
    return null;  
  }  
  const parts = dateOfBirth.split('-');  
  return new Date(+parts[0], +parts[1] - 1, +parts[2]);  
}
```

6.2 Implement Separate 2 Front End Applications

1. Overview

The implementation in 6.1 has only one Angular application which include features for both user management and course management. Now to do the POC of separate FE applications, we are going to split the current Angular application course-ap into to applications:

- user-app: User Management App
 - Configured with own OKTA client Id and redirect URL
 - Redirects to user management dashboard after login, which is the redirect URL on OKTA for User Management App
- course-app: Course Management App
 - Configured with own OKTA client Id and redirect URL
 - Redirects to course management dashboard after login, which is the redirect URL on OKTA for Course Management App
- OKTA
 - Single ORG

Target:

- 2 separately deployed Angular applications
- SSO between apps still works
- Backend cross-app communication still works
- Able to redirect to a specified page in the other application other than default page (Dashboard)

2. Implementation

2.1 Front End

Step 1: Create a Workspace with Multiple Projects

The current **course-app** is a **single-project** Angular app, first convert it into a workspace that **supports multiple apps and libraries**.

```
cd D:\workspace\sso-demo\sso-okta-app-authorization\frontend
ng new fe-workspace --create-application=false
cd fe-workspace
```

Add the existing course-app

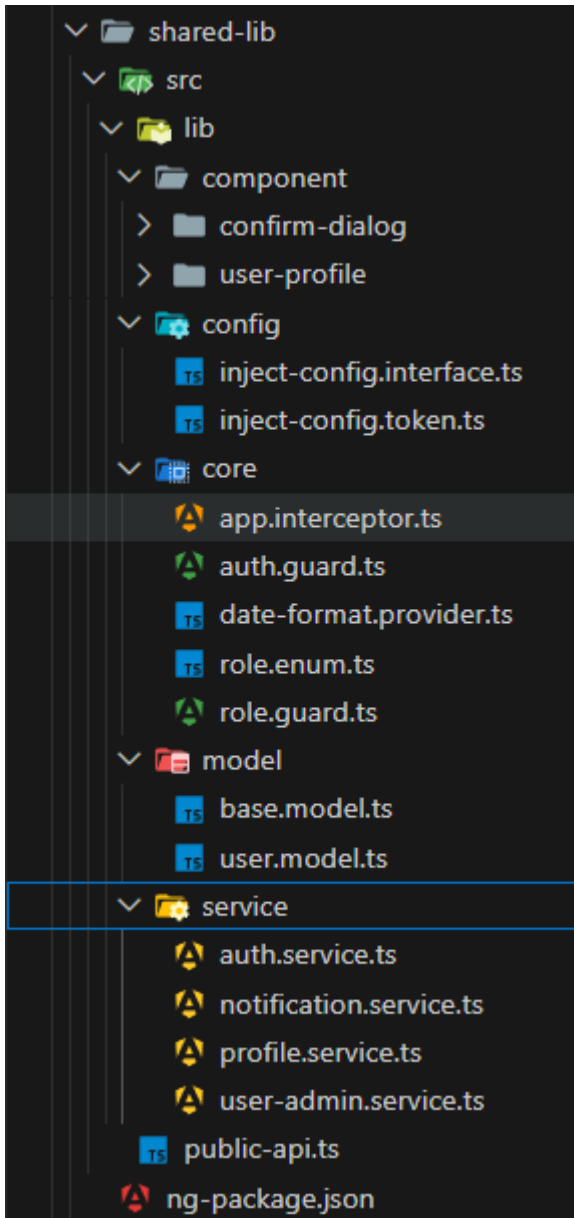
```
ng generate application course-app
```

Step 2: Create user-app and shared-lib

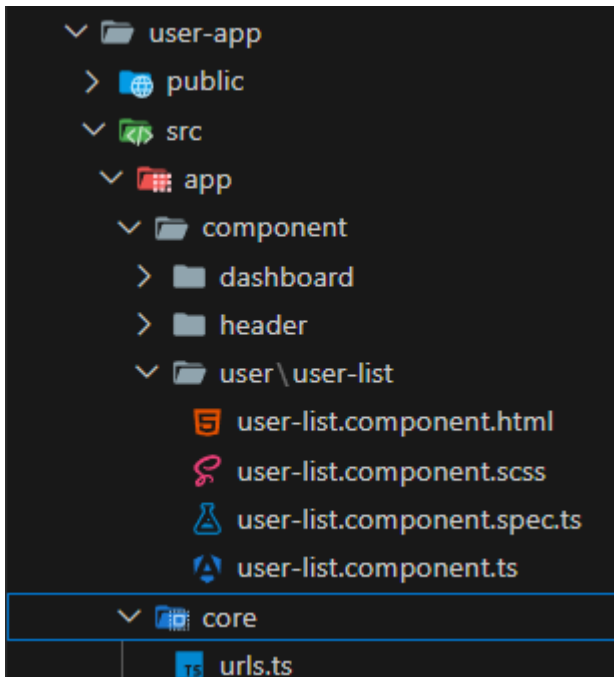
```
ng generate application user-app  
ng generate library shared-lib
```

Step 3: Move Shared Code to shared-lib

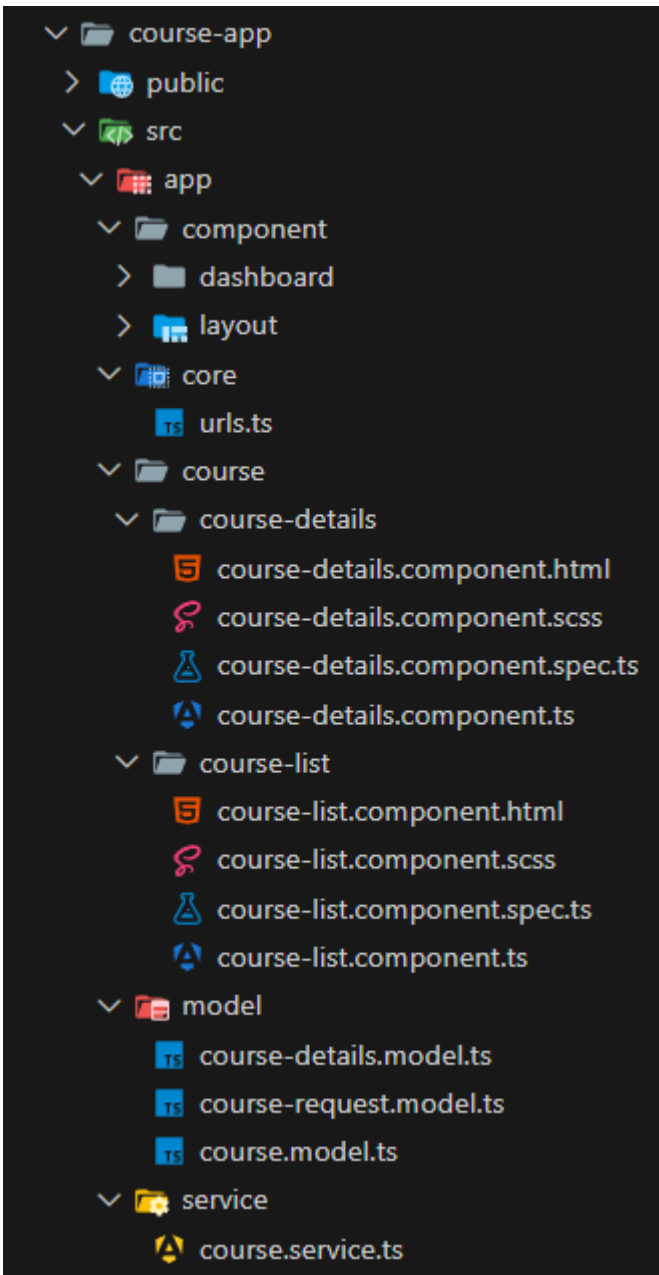
To update when finalize



Step 4: Move User-Related Features to user-app



Step 5: Move Course-Related Features to course-app



Step 6: Import shared-lib in Both Apps

Make sure **fe-workspace/tsconfig.json** includes the path alias for the library:

```
"paths": {
  "shared-lib": ["projects/shared-lib/src/public-api"]
}
```

and **tsconfig.app.json** in both user-app and course-app extends from it:

```
{
  "extends": "../../tsconfig.json",
  ...
}
```

Step 7: Import shared-lib in Both Apps

- Ensure both course-app and user-app have **independent routing**
- Each can have its own **proxy.conf.json** to call backend APIs through the gateway

Step 8: External Configuration Injection

1. Define an Interface for Config Data

```
export interface SharedLibConfig {
  loginUrl: string;
  profileApiUrl: string;
  usersApiUrl: string;
}
```

2. Create an Injection Token

```
import { InjectionToken } from '@angular/core';
import { SharedLibConfig } from './shared-lib-config.interface';

export const SHARED_LIB_CONFIG = new InjectionToken<SharedLibConfig>('SharedLibConfig');
```

3. Inject It into a Service or Component in the shared-lib, e.g. use it in AuthGuard

```
import { inject } from '@angular/core';
import { CanActivateFn } from '@angular/router';
import { tap } from 'rxjs/operators';
import { AuthService } from '../service/auth.service';
import { SHARED_LIB_CONFIG } from '../config/shared-lib-config.token';
import { SharedLibConfig } from '../config/shared-lib-config.interface';

export const AuthGuard: CanActivateFn = () => {

  const sharedLibConfig: SharedLibConfig = inject(SHARED_LIB_CONFIG);
  const authService = inject(AuthService);

  return authService.checkAuthentication().pipe(
    tap(isAuthenticated => {
      if (!isAuthenticated) {
        window.location.href = sharedLibConfig.loginUrl;
      }
    })
  );
};
```

4. Provide It from the Host App

```
import { routes } from './app.routes';
import { AppInterceptor, DateFormatProvider, SHARED_LIB_CONFIG, SharedLibConfig } from 'shared-lib';
import { URL_LOGIN, URL_PROFILE_API, URL_USERS_API } from './core/urls';

const sharedLibConfig: SharedLibConfig = {
  loginUrl: URL_LOGIN,
  profileApiUrl: URL_PROFILE_API,
  usersApiUrl: URL_USERS_API
}

export const appConfig: ApplicationConfig = {
  providers: [provideZoneChangeDetection({ eventCoalescing: true }),
    provideHttpClient(withInterceptorsFromDi()), // ✅ This fixes HttpClient injection
    ... ..
    // Inject config to the shared lib
    { provide: SHARED_LIB_CONFIG, useValue: sharedLibConfig }
  ]
};
```

Step 9: Ensure Run Auth Check Before Load Page when Switch to Another App

❗ **Issue:** When switch to a page other than default page in the other application, it always land to default page Dashboard.

✅ What's Happening

When you switch from **user-app** to **course-app**, the following sequence happens:

1. Your browser navigates to `http://localhost:4201/courses`.
2. Angular app initializes.
3. Angular route guard runs for `/courses` → uses `AuthService.isAuthenticated$`.
4. **BUT:** `checkAuthentication()` hasn't completed yet (or hasn't even started).
5. So the guard treats the user as **unauthenticated** → **redirects to home or dashboard**.
6. Meanwhile, **`checkAuthentication()`** runs and confirms authentication — but it's **too late**.

✅ Why This Happens

It's calling `checkAuthentication()` in `AppComponent.ngOnInit()`, which:

- Is **not guaranteed to complete before routing starts**
- Doesn't synchronously block the route guard

✅ Clean Solution

Modify your AuthGuard to:

- **Call `checkAuthentication()` directly** inside `canActivate()` (see above updated code)
- Wait for the result (`Observable<boolean>`)

- Only allow navigation **after** the async check finishes

auth.service.ts: change to return Observable<boolean>

```

export class AuthService {

  private authenticated = new BehaviorSubject<boolean>(false);
  public isAuthenticated$ = this.authenticated.asObservable();

  private _profile!: User | null;

  public get profile() {
    return this._profile;
  }

  constructor(private profileService: ProfileService) {}

  private checkInProgress$: Observable<boolean> | null = null;

  checkAuthentication(): Observable<boolean> {
    // ✅ If already authenticated, return true immediately
    if (this.authenticated.value === true) {
      return of(true);
    }

    // ✅ If already checking, return the same observable
    if (this.checkInProgress$) {
      return this.checkInProgress$;
    }

    // 🔄 Otherwise, perform check and cache it
    const check$ = this.profileService.getProfile().pipe(
      tap(user => {
        this._profile = user;
        this.authenticated.next(true);
      }),
      map(() => true),
      catchError((error: HttpErrorResponse) => {
        if (error.status === 404) {
          // Try creating profile
          return this.profileService.createProfile().pipe(
            tap(createdUser => {
              this._profile = createdUser;
              this.authenticated.next(true);
            }),
            map(() => true),
            catchError(err => {
              console.error('❌ Failed to create profile', err);
              this._profile = null;
              this.authenticated.next(false);
              return of(false);
            })
          );
        } else {
          console.error('❌ Failed to load profile', error);
          this._profile = null;
          this.authenticated.next(false);
          return of(false);
        }
      })
    ),
    finalize(() => {
      // 🔄 Clear cache after first run completes
      this.checkInProgress$ = null;
    })
  );
}

```

```

    this.checkInProgress$ = check$;
    return check$;
  }
}

```

Call `authService.checkAuthentication()` manually in `header.component`

Also, call the REST API `"/custom-login/{client}?redirectTo="` for redirecting to specified page

```

isAuthenticated$ = this.authService.isAuthenticated$;

ngOnInit(): void {
  this.authService.checkAuthentication()
    .pipe(takeUntil(this.destroy$))
    .subscribe();
}

redirectToCourseApp(targetPath: string): string {
  const encodedRedirectTo = encodeURIComponent(targetPath);
  return `http://localhost:9001/custom-login/course-app?redirectTo=${encodedRedirectTo}`;
}

```

Remove `authService.checkAuthentication()` in `app.component`.

Step 10: Misc Changes

- Add necessary dependencies in `package.json`

```

"@angular/cdk": "^19.0.0"
"@angular/material": "^19.2.17"
"bootstrap": "^5.3.6"
"date-fns": "^4.1.0"

```

- Add link resource for icons in `index.html` of both applications

```

<link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet" />

```

- export necessary artifacts in `public-api.ts`

```
/*  
 * Public API Surface of shared-lib  
 */  
  
export * from './lib/model/base.model';  
export * from './lib/model/user.model';  
  
export * from './lib/component/confirm-dialog/confirm-dialog.component';  
... ..
```

2.2 Backend

To support SSO between two **separately hosted Angular apps (on ports 4200 and 4201)** with two **backend OKTA clients (user-app and course-app)**, you must ensure that after logging into user-app (<http://localhost:4200>), you should be able to access course-app (<http://localhost:4201>) **without** logging in again — and vice versa.

2.2.1 Update api-gateway SecurityConfig to support multiple clients

application.xml of **api-gateway**

```
client:  
  redirect-url:  
    user-app: http://localhost:4200  
    course-app: http://localhost:4201
```

SecurityConfig.java

```

@Configuration
@EnableWebFluxSecurity
public class SecurityConfig {

    @Value("${client.redirect-url.user-app}")
    private String userAppRedirectUrl;

    @Value("${client.redirect-url.course-app}")
    private String courseAppRedirectUrl;

    @Bean
    public SecurityWebFilterChain springSecurityFilterChain(
        ServerHttpSecurity http,
        ReactiveClientRegistrationRepository clientRegistrationRepository) {

        return http
            .csrf(ServerHttpSecurity.CsrfSpec::disable)
            .cors(cors -> cors.configurationSource(corsConfigurationSource()))
            .authorizeExchange(exchanges -> exchanges
                .pathMatchers("/actuator/**", "/public/**").permitAll()
                .anyExchange().authenticated())
            .oauth2Login(login -> login
                .authenticationSuccessHandler(redirectToAngular())) // redirects to Angular after
Login
            .logout(logout -> logout
                .logoutSuccessHandler(oidcLogoutSuccessHandler(clientRegistrationRepository)))
            .build(); // ✅ TokenRelay is configured via application.yml route filters
    }

    private ServerAuthenticationSuccessHandler redirectToAngular() {
        return (exchange, authentication) -> {
            ServerWebExchange webExchange = exchange.getExchange();

            String defaultRedirect = getClientRedirectUrl((OAuth2AuthenticationToken) authentication);

            return webExchange.getSession().flatMap(session -> {
                String redirectTo = (String) session.getAttributes().get("redirectTo");
                log.info("Redirecting to {}", redirectTo);

                // Clear it from session after use
                session.getAttributes().remove("redirectTo");

                String finalRedirect = (redirectTo != null && !redirectTo.isBlank())
                    ? defaultRedirect + redirectTo
                    : defaultRedirect;

                RedirectServerAuthenticationSuccessHandler handler =
                    new RedirectServerAuthenticationSuccessHandler(finalRedirect);

                return handler.onAuthenticationSuccess(exchange, authentication);
            });
        };
    }

    private ServerLogoutSuccessHandler oidcLogoutSuccessHandler(
        ReactiveClientRegistrationRepository clientRegistrationRepository) {

        return (exchange, authentication) -> {
            String redirectUri = getClientRedirectUrl((OAuth2AuthenticationToken) authentication);

```

```

        new OidcClientInitiatedServerLogoutSuccessHandler(clientRegistrationRepository);
        handler.setPostLogoutRedirectUri(String.valueOf(URI.create(redirectUri)));

        return handler.onLogoutSuccess(exchange, authentication);
    };
}

private String getClientRedirectUrl(OAuth2AuthenticationToken authentication) {
    String client = authentication.getAuthorizedClientRegistrationId();

    return switch (client) {
        case "user-app" -> userAppRedirectUrl;
        case "course-app" -> courseAppRedirectUrl;
        default -> userAppRedirectUrl; // fallback
    };
}

@Bean
public UrlBasedCorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration config = new CorsConfiguration();
    config.setAllowedOrigins(List.of("http://localhost:4200", "http://localhost:4201"));
    config.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE", "OPTIONS"));
    config.setAllowedHeaders(List.of("*"));
    config.setAllowCredentials(true); //  important for cookies/session-based auth

    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", config);
    return source;
}
}

```

2.2.2 Create a Custom Controller to Initiate Login

To reliably capture `redirectTo`, we should **not use** the default Spring Security endpoint `/oauth2/authorization/{registrationId}` directly.

Instead, create your own endpoint (e.g. `/custom-login/course-app`) that:

1. Reads the `redirectTo` query param,
2. Stores it in the session,
3. Redirects to the real Spring Security OAuth2 login URL (`/oauth2/authorization/course-app`).

OAuth2LoginController.java

```

@RestController
@RequestMapping("/custom-login")
public class OAuth2LoginController {

    @GetMapping("/{client}")
    public Mono<Void> loginWithRedirect(@PathVariable String client,
                                       @RequestParam(required = false) String redirectTo,
                                       ServerWebExchange exchange) {
        return exchange.getSession().flatMap(session -> {
            if (redirectTo != null) {
                session.getAttributes().put("redirectTo", redirectTo);
            }

            String loginUri = "/oauth2/authorization/" + client;

            return Mono.defer(() -> {
                exchange.getResponse().setStatusCode(HttpStatus.FOUND);
                exchange.getResponse().getHeaders().setLocation(URI.create(loginUri));
                return exchange.getResponse().setComplete();
            });
        });
    }
}

```

Use This Controller in the Frontend Links

header.component.ts

```

redirectToCourseApp(targetPath: string): string {
    const encodedRedirectTo = encodeURIComponent(targetPath);
    return `http://localhost:9001/custom-login/course-app?redirectTo=${encodedRedirectTo}`;
}

```

7. Architecture and Design

1. Architecture

1.1 Architecture Options

1.1.1 Should we use PKCE?

OKTA recommends PKCE only for **public clients** (e.g., SPAs or mobile apps).

Even though our frontend is an SPA:

- The **Angular app never talks to Okta directly**
- All OAuth2 traffic goes **through the Spring Boot Gateway**
- And that gateway uses a **client ID + secret = confidential client**

The backend does not expose credentials to the browser, so **PKCE** is **NOT recommended** in our case.

1.1.2 Communication between FE and BE

Expose only the gateway to the public.

All frontend (browser) requests should go through the **gateway** (Spring Cloud Gateway).

All microservices like user-profile-svc, course-query-svc, etc. should:

- Be **behind the gateway**
- Be accessible **only inside the internal network**

Currently for POC purpose, we are using 2 separate gateways (SCG). Given our application, the actual implementation should use **SINGLE** gateway.

1.1.3 Inter-Service (Microservices) Communication

- It currently includes the user's access token in the **Authorization: Bearer ... header**
- user-profile-svc accepts that **token** via Spring Security's **oauth2ResourceServer.jwt** config

✅ What's Protected

- The **user's identity and token** is validated at user-profile-svc
- The API is not open to the public without a valid OKTA token

❌ What's Not Protected

- Anyone on the same network (e.g., compromised internal machine or Docker container) could call with a valid OKTA access token and get data — because there's no **caller identity verification**

● Recommendation for You (Today)

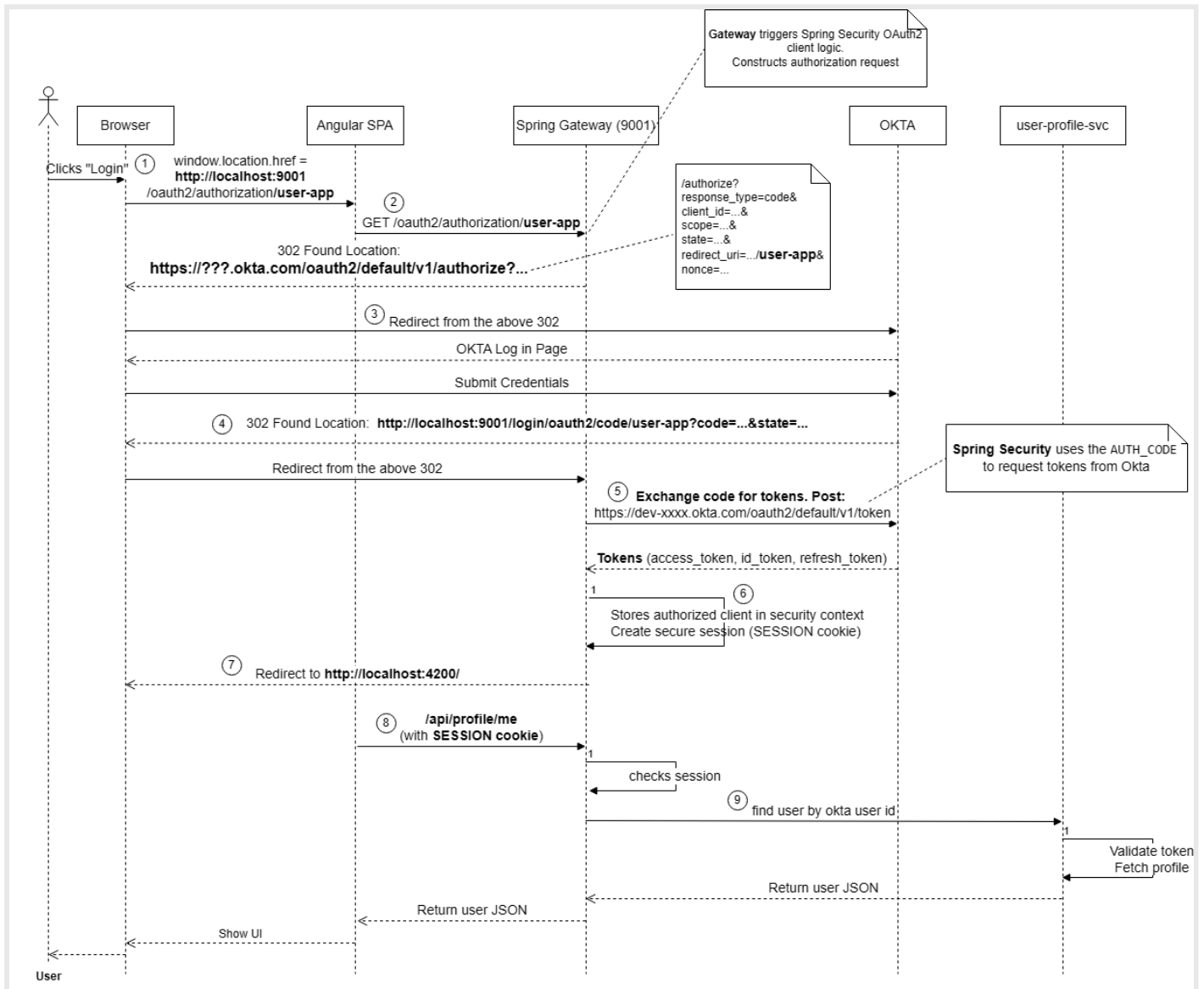
Since your goal is **app-managed roles and backend-only access**, here's what's ideal right now:

Protection Type	Recommended Now?	Notes
Private network	✅ Must-have	Use internal-only ports/IPs

Token validation	✓ Already used	Validates user's identity
Caller identity check	✓ Add header check	Fast and simple
mTLS / service JWTs	➔ SOON Optional later	Use when hardening prod

2. Design

2.1 Login Sequence Diagram



2.1.1 Actors Involved:

- **User:** A person interacting with the app.
- **Browser:** Executes the Angular SPA and handles HTTP redirects.
- **Angular SPA:** Frontend app running in the browser on port 4200.
- **Spring Cloud Gateway:** Backend reverse proxy with OAuth2 login configured on port 9001.
- **Okta:** The identity provider handling login and token issuance.

2.1.2 Full Authentication Sequence (Authorization Code Flow with PKCE)

1. User initiates login

- **User** clicks “Login” in the Angular app.
- **Angular SPA** triggers:
→ window.location.href = <http://localhost:9001/oauth2/authorization/user-app>

2. Redirect to Okta for login

- **Browser** sends GET /oauth2/authorization/user-app to **Gateway**
- **Gateway** triggers Spring Security OAuth2 client logic:
 - Constructs authorization request
 - Redirects to:

```
https://dev-50623690.okta.com/oauth2/default/v1/authorize?  
response_type=code&  
client_id=...&  
scope=...&  
state=...&  
redirect_uri=http://localhost:9001/login/oauth2/code/user-app&  
nonce=...
```

3. User authenticates with Okta

- **Browser** follows redirect to Okta (See the above 302 location).
- **User** enters username and password on the **Okta login page**
- If MFA is enabled, **User** completes MFA (e.g. Okta Verify)
- **Okta** validates credentials and MFA

4. Okta returns authorization code

. Okta redirects back to <http://localhost:9001/login/oauth2/code/user-app?code=...&state=...>

5. Spring Gateway exchanges code for tokens

- **Gateway** receives the request on /login/oauth2/code/user-app
- **Spring Security** uses the AUTH_CODE to request tokens from Okta:
 - Spring Security captures the redirect (/login/oauth2/code/user-app)
 - extracts the code=... from the URL
 - sends a **POST** request to Okta's token endpoint

- POST `https://your-domain.okta.com/oauth2/default/v1/token`
Content-Type: `application/x-www-form-urlencoded`

```
grant_type=authorization_code
code=abc123...
redirect_uri=http://localhost:9001/login/oauth2/code/okta
client_id=...
client_secret=...
```

- If valid, Okta responds with:

```
{
  "access_token": "...",
  "id_token": "...",
  "expires_in": 3600,
  "token_type": "Bearer"
}
```

- Spring stores these tokens (usually in session)
- User is now authenticated

6. Session established on Gateway

- **Gateway** stores the authorized client in the security context
- A secure session (SESSION cookie) is created


7. Gateway redirects user back to Angular

- **Gateway** redirects user to configured success URL


<http://localhost:4200> 

8. Angular bootstraps and requests user profile

- **Angular SPA** loads
- It calls:
→ GET `/api/profile/me` via proxy (→ `http://localhost:9001/api/profile/me`)
- **Gateway** checks session, forwards request with access token to backend
 - **Spring Security inspects the session cookie (SESSION):**
 - This session was created during login (`/login/oauth2/code/user-app`).
 - The session holds the **OAuth2AuthorizedClient** object, which contains:
 - Access token
 - Refresh token (optional)
 - Client registration metadata
 - The session is **authenticated**, so request proceeds.
 - Because we added the filter **"TokenRelay"**,

- **SCG** retrieves the **access token** from the **OAuth2AuthorizedClient** in the session and,
- Adds header: **Authorization: Bearer eyJhbGciOi...**
- **SCG forwards the request** to the backend microserviceservice discovery (via **lb://...** in uri .

9. Spring Boot microservice validates token, fetches profile, and returns JSON

 Incoming Request to microservice:

```
GET /api/profile/me
Authorization: Bearer eyJhbGciOi...
```

- We have configured the service as a **JWT resource server**:

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

- **Spring Security validates the token:**
 - **Decodes** the **JWT**
 - **Checks claims** like:
 - **iss** (issuer)
 - **exp** (expiration)
 - **aud** (audience, if configured)
 - If the token is valid:
 - Spring populates the **Authentication** object with a **JwtAuthenticationToken**
 - In Java code, we can access claims via:

```
@AuthenticationPrincipal Jwt jwt
```

- controller or service logic uses the claim (e.g., email) to fetch the user profile

```
AppUser user = userRepository.findByEmail(jwt.getClaimAsString("email"));
```

2.2 Design Approach for Separate Schemas

2.2.1 Tables

1. course – in schema **ssocourse**
2. app_user – in schema **ssouser**

3. course_enrollment – join table in **ssocourse** schema

♦ course-enrollment table structure

Column	Type	Description
course_id	BIGINT	FK to course.id
student_id	BIGINT	FK to app_user.id (role=STUDENT)
created_at	TIMESTAMP	optional, for audit

⚠ No actual FK constraint on **student_id** since it's cross-schema — handle integrity in application logic.

⚠ **Do not share entities across services** unless they use the same schema. Misusing might cause the same entity creates DB table in multiple databases (schemas)

In our case, we have the following entities created in common-lib:

- `com.dxu.sso.common.model.user.AppUser`
- `com.dxu.sso.common.model.course.Course`

We create them in different packages, then we can scan them separately in necessary microservices

For inter-service communication between services for different databases/schemas, use DTO instead