페더레이션 **ID** 및 중앙 집중식 권한 부여 허브 아키텍처 구축

서론**:** 페더레이션 **ID** 및 권한 부여 허브 아키텍처 설계

현대적인 분산 시스템 환경에서 일관되고 안전하며 확장 가능한 인증 및 권한 부여 메커니즘을 구축하는 것은 매우 중요한 과제입니다. 본 보고서는 외부 ID 공급자(Identity Provider, IdP)와의 연동을 통해 인증을 위임하고, 내부 비즈니스 로직에 따라 동적으로 권한을 부여하여 자체 서명된 JWT(JSON Web Token)를 발급하는 중앙 집중식 "ID 허브(Identity Hub)" 아키텍처의 설계 및 구현 방안을 상세히 기술합니다. 이 아키텍처의 핵심은 인증(Authentication) 프로세스와 권한 부여 토큰 발급(Authorization Token Issuance) 프로세스를 전략적으로 분리하는 데 있습니다.

핵심 개념**: ID** 허브 아키텍처

ID 허브는 단순히 외부 IdP에 대한 프록시 역할을 넘어, 조직의 모든 애플리케이션에 대한 인증 및 권한 부여 정책을 중앙에서 관리하고 제어하는 전략적 자산으로 기능합니다. 외부 IdP(본 보고서에서는 Azure Active Directory, 현재 Microsoft Entra ID)는 사용자의 신원을 확인하는 역할만 담당하며, 인증이 성공적으로 완료되면 ID 허브는 이 사용자에 대한 내부 비즈니스 컨텍스트(예: 데이터베이스에 저장된 사용자 역할)를 결합하여 새로운 JWT를 생성합니다. 이 JWT는 내부 애플리케이션 생태계에서 사용되는 표준화된 자격 증명 역할을 합니다.

해결 과제

이 아키텍처는 마이크로서비스 환경에서 흔히 발생하는 다음과 같은 문제점들을 해결합니다.

- 분산된 권한 부여 로직**:** 각 마이크로서비스가 개별적으로 권한 부여 로직을 구현함으로써 발생하는 일관성 부족 및 관리의 복잡성.

- 특정 **IdP** 종속성**:** 애플리케이션이 특정 IdP(예: Azure AD)의 토큰 형식이나 클레임 구조에 강하게 결합되어, 향후 IdP 변경 시 막대한 수정 비용이 발생하는 문제.
- 컨텍스트 부족 토큰**:** 외부 IdP가 발급한 토큰에는 내부 시스템에서 필요한 비즈니스 관련 정보(예: 사용자 등급, 부서별 권한)가 포함되지 않아 각 서비스가 추가적인 데이터 조회를 수행해야 하는 비효율성.

솔루션 개요

본 보고서에서 제시하는 솔루션의 전체적인 데이터 흐름은 다음과 같습니다.

1. 사용자 인증 요청**:** 사용자가 Nuxt.js 웹 클라이언트 또는.NET WinForms 데스크톱 클라이언트를 통해 로그인을 시도합니다.
2. 인증 위임**:** 클라이언트는 중앙 ID 허브(커스텀 Spring Authorization Server)로 리디렉션됩니다. ID 허브는 OIDC(OpenID Connect) 프로토콜을 사용하여 사용자를 다시 Azure AD로 리디렉션하여 실제 인증을 위임합니다.
3. 외부 인증 및 콜백**:** 사용자는 Azure AD에서 성공적으로 인증하고, ID 허브는 콜백을 통해 인증 결과를 전달받습니다.
4. 토큰 강화**(Enrichment):** ID 허브는 Azure AD로부터 받은 ID 토큰에서 사용자 식별 정보를 추출한 후, 내부 데이터베이스를 조회하여 해당 사용자의 역할(Role) 목록을 가져옵니다.
5. 커스텀 **JWT** 발급**:** ID 허브는 Azure AD의 기본 사용자 정보와 내부 DB의 역할 정보를 결합하여 새로운 페이로드(Payload)를 구성하고, 자체 개인 키로 서명한 새로운 JWT를 생성하여 클라이언트에게 발급합니다.
6. 리소스 접근**:** 클라이언트는 발급받은 커스텀 JWT를 Authorization: Bearer 헤더에 담아 리소스 서버(Spring Boot & Thymeleaf)의 보호된 API를 호출합니다.
7. 토큰 검증 및 권한 부여**:** 리소스 서버는 ID 허브의 공개 키를 사용하여 수신된 JWT의 서명을 검증하고, 토큰 내의 역할(roles) 클레임을 기반으로 요청된 작업에 대한 접근을 허용하거나 거부합니다.

코드 스니펫

```
@startuml
actor User
participant "Client App\n(Nuxt.js / WinForms)" as Client
participant "ID Hub\n(Spring Auth Server)" as AuthServer
participant "Azure AD\n(External IdP)" as AzureAD
database "Internal DB" as DB
participant "Resource Server" as ResourceServer
```

```
User -> Client: 1. 로그인 시도
Client -> AuthServer: 2. 인증 요청 (/oauth2/authorize)
activate AuthServer
AuthServer -> AzureAD: 3. OIDC 인증 위임
deactivate AuthServer
User -> AzureAD: 4. Azure AD 로그인
AzureAD --> User: 인증 성공
AzureAD -> AuthServer: 5. Authorization Code와 함께 콜백
activate AuthServer
AuthServer -> AzureAD: 6. Code를 Token으로 교환 요청
activate AzureAD
AzureAD --> AuthServer: 7. ID Token, Access Token, Refresh Token 응답
deactivate AzureAD
AuthServer -> DB: 8. 사용자 역할(Role) 정보 조회
activate DB
DB --> AuthServer: 역할 정보 응답
deactivate DB
AuthServer -> AuthServer: 9. ID Token 정보와 역할 정보를 결합하여\n  자체 JWT(Access Token) 생성
AuthServer -> Client: 10. 자체 JWT와 함께 리디렉션
deactivate AuthServer

... 잠시 후...

User -> Client: 보호된 리소스 요청
Client -> ResourceServer: 11. API 호출 (Authorization: Bearer)
activate ResourceServer
ResourceServer -> AuthServer: 12. 공개키(JWKS) 요청 (최초 1회)
activate AuthServer
AuthServer --> ResourceServer: 공개키 응답
deactivate AuthServer
ResourceServer -> ResourceServer: 13. JWT 서명 검증 및\n  'roles' 클레임 확인
ResourceServer --> Client: 14. 보호된 리소스 응답
deactivate ResourceServer
Client --> User: 결과 표시
@enduml
```

전략적 이점

이러한 ID 허브 패턴은 다음과 같은 장기적인 이점을 제공합니다.

- 중앙 집중식 권한 부여**:** 모든 역할 기반 접근 제어(RBAC) 로직이 ID 허브 한 곳에서 관리되므로, 정책 변경이 용이하고 전체 시스템의 일관성이 보장됩니다.
- **IdP** 독립성**(Agnosticism):** 내부 리소스 서버 및 클라이언트는 ID 허브가 발급한 표준 JWT만 인식합니다. 따라서 향후 외부 IdP를 Azure AD에서 Okta나 Google로 변경하더라도 내부 시스템에는 아무런 영향을 미치지 않습니다.[1]
- 컨텍스트 인식 토큰**:** JWT 자체에 비즈니스 컨텍스트(역할, 등급 등)가 포함되므로, 리소스 서버는 추가적인 DB 조회 없이 토큰만으로 권한 부여를 수행할 수 있어 구조가 단순해지고 성능이 향상됩니다.[2]
- 일관된 보안 모델**:** 생태계 내의 모든 애플리케이션이 단일한 토큰 형식과 검증 메커니즘을 따르므로, 보안 모델이 통일되고 예측 가능해집니다.

이 아키텍처의 핵심은 인증 행위(Azure AD에 위임)와 권한 부여 토큰 발급 행위(자체 서버가 담당)를 명확히 분리하는 것입니다. 자체 인증 서버는 외부의 '인증된 주체(Authenticated Principal)'를 내부 시스템이 이해할 수 있는 '권한을 부여받은 주체(Authorized Principal)'로 변환하는 번역기 역할을 수행합니다. 이 변환 계층은 내부 애플리케이션들을 외부 IdP의 구체적인 구현(토큰 구조, 클레임, 엔드포인트 등)으로부터 완벽하게 분리시키는 강력한 추상화 계층이며, 이는 시스템의 유지보수성과 미래 확장성을 극대화하는 중요한 아키텍처 결정입니다.

Part I: Spring Boot Authorization Server: 페더레이션 및 토큰 강화

ID 허브의 핵심인 Spring Boot 기반의 커스텀 Authorization Server를 구축하는 과정을 상세히 설명합니다. 이 서버는 외부 IdP인 Azure AD와의 연동, 내부 데이터베이스를 통한 사용자 역할 정보 조회, 그리고 이 두 정보를 결합한 커스텀 JWT를 발급하는 책임을 가집니다.

섹션 **1.1:** 기반 구축 및 설정

프로젝트 초기화 및 의존성 분석

먼저 Spring Initializr를 사용하여 새로운 Spring Boot 프로젝트를 생성합니다. 프로젝트 생성 시 다음 의존성들을 반드시 포함해야 합니다.[3]

- **Spring Authorization Server:** OAuth 2.1 및 OpenID Connect 1.0 사양의 구현을 제공하는 핵심 프레임워크입니다.[4]
- **Spring Web:** REST 컨트롤러 및 웹 애플리케이션 구성을 위해 필요합니다.
- **Spring Security:** 인증 및 권한 부여의 기반을 제공합니다.
- **Spring Data JPA:** 데이터베이스와의 상호작용을 위한 ORM(Object-Relational Mapping)을 제공합니다.
- **H2 Database:** 개발 및 테스트 환경을 위한 인메모리 데이터베이스입니다.
- **Oracle Driver:** 운영 환경을 위한 Oracle 데이터베이스 드라이버입니다.
- **Spring Boot Actuator:** 애플리케이션의 모니터링 및 관리를 위한 엔드포인트를 제공합니다.

각 의존성은 명확한 역할을 가집니다. Spring Authorization Server는 OAuth 2.1 프로토콜 스택을 제공하여 클라이언트 등록, 토큰 발급 등의 기능을 담당합니다. Spring Data JPA는 사용자 역할 정보를 저장하는 비즈니스 데이터베이스뿐만 아니라, OAuth2 클라이언트 설정과 사용자 권한 부여 상태를 영속적으로 관리하는 데에도 사용됩니다.

클라이언트 등록**:** 메모리 기반 설정

요청에 따라, 인증 서버에 접속하는 클라이언트 정보는 데이터베이스가 아닌 application.yml 파일에 직접 설정합니다. 이를 위해 InMemoryRegisteredClientRepository를 사용합니다. 이 방식은 클라이언트 수가 적고 정적인 환경에 적합합니다.[5]

Java

```
@Bean
public RegisteredClientRepository registeredClientRepository() {
    // Nuxt.js 클라이언트 (Public Client, PKCE 사용)
    RegisteredClient nuxtClient = RegisteredClient.withId(UUID.randomUUID().toString())
        .clientId("nuxt-client")
        .clientAuthenticationMethod(ClientAuthenticationMethod.NONE)
        .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
        .redirectUri("http://localhost:3000/api/auth/callback") // Nuxt 서버 콜백
```

```
        .postLogoutRedirectUri("http://localhost:3000/")
        .scope(OidcScopes.OPENID)
        .scope(OidcScopes.PROFILE)
        .clientSettings(ClientSettings.builder().requireProofKey(true).build()) // PKCE 활성화
        .build();

    // WinForms 클라이언트 (Public Client, PKCE 사용)
    RegisteredClient winformsClient = RegisteredClient.withId(UUID.randomUUID().toString())
        .clientId("winforms-client")
        .clientAuthenticationMethod(ClientAuthenticationMethod.NONE)
        .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
        .redirectUri("http://127.0.0.1:7890/callback") // 루프백 주소
        .scope(OidcScopes.OPENID)
        .scope(OidcScopes.PROFILE)
        .clientSettings(ClientSettings.builder().requireProofKey(true).build()) // PKCE 활성화
        .build();

    return new InMemoryRegisteredClientRepository(nuxtClient, winformsClient);
}
```

데이터베이스 스키마 및 환경별 설정

Spring Authorization Server는 외부 IdP로부터 받은 토큰(Access Token, Refresh Token)을 저장하여 SSO 세션을 유지해야 합니다. 이 정보는 oauth2_authorized_client 테이블에 저장됩니다. 이를 위해 JdbcOAuth2AuthorizedClientService를 사용합니다.[8]

개발 환경(dev)에서는 H2 인메모리 데이터베이스를, 운영 환경(prod)에서는 Oracle 데이터베이스를 사용하도록 Spring Profiles를 활용하여 설정을 분리합니다.[9]

src/main/resources/application.yml (공통 설정)

YAML

```
spring:
  profiles:
    active: dev # 기본 프로파일을 dev로 설정
  jpa:
```

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

src/main/resources/application-dev.yml (개발 환경: H2)

YAML

```
spring:
  datasource:
    url: jdbc:h2:mem:authdb
    driver-class-name: org.h2.Driver
    username: sa
    password: password
  jpa:
    database-platform: org.hibernate.dialect.H2Dialect
  h2:
    console:
      enabled: true
      path: /h2-console
```

src/main/resources/application-prod.yml (운영 환경: Oracle)

YAML

```
spring:
  datasource:
    url: jdbc:oracle:thin:@your-oracle-host:1521:your-sid
    username: your-username
    password: your-password
    driver-class-name: oracle.jdbc.OracleDriver
  jpa:
    database-platform: org.hibernate.dialect.Oracle12cDialect
    hibernate:
      ddl-auto: validate # 운영에서는 스키마 자동 생성을 사용하지 않음
```

JWT는 종종 상태 비저장(stateless) 아키텍처와 연관되지만, 외부 IdP와 연동하는 본

아키텍처에서는 필연적으로 상태를 관리하는 요소가 필요합니다. oauth2_authorized_client 테이블이 바로 그 상태 저장소입니다. 이 테이블은 Azure AD로부터 받은 토큰들을 저장하며, 이 상태 정보가 있어야만 사용자가 다시 로그인하지 않고도 ID 허브가 Azure AD와의 세션을 갱신할 수 있습니다. 즉, 이 아키텍처는 하이브리드 모델을 채택합니다. ID 허브와 Azure AD 간의 관계는 상태 기반의 세션으로 관리되며, ID 허브와 내부 리소스 서버들 간의 관계는 자체 포함된(self-contained) JWT를 통해 상태 비저장 방식으로 운영됩니다. 이 구조적 특징을 이해하는 것은 전체 시스템의 동작 방식을 파악하는 데 매우 중요합니다.[8]

섹션 **1.2: Azure Active Directory**와의 페더레이션

ID 허브가 사용자를 인증하기 위해 Azure AD와 연동하는 과정을 설정합니다.

Azure Portal 설정

Azure Portal에서 애플리케이션을 등록하는 과정은 다음과 같습니다.[14]

1. **Microsoft Entra ID(Azure AD) >** 앱 등록으로 이동하여 '새 등록'을 선택합니다.
2. 애플리케이션의 이름을 입력하고 지원되는 계정 유형을 선택합니다.
3. 리디렉션 **URI**를 설정합니다. 플랫폼으로 '웹'을 선택하고, URI는 Spring Security OAuth2 클라이언트의 기본 콜백 주소인 {baseUrl}/login/oauth2/code/{registrationId} 형식으로 입력합니다. 예를 들어, 로컬 환경에서 실행하고 등록 ID를 'azure'로 사용할 경우 http://localhost:9000/login/oauth2/code/azure가 됩니다.[16]
4. 애플리케이션이 등록되면 '개요' 페이지에서 애플리케이션**(**클라이언트**) ID**와 디렉터리**(**테넌트**) ID**를 기록해 둡니다.
5. 인증서 및 암호 메뉴로 이동하여 '새 클라이언트 암호'를 생성하고, 생성된 **값(Value)**을 안전한 곳에 즉시 복사합니다. 이 값은 다시 볼 수 없습니다.

Spring Boot 설정 **(application.yml)**

Azure Portal에서 얻은 정보를 사용하여 application.yml 파일에 OIDC 클라이언트를 등록합니다. spring-cloud-azure-starter-active-directory를 사용하면 일부 설정이 간소화될 수 있지만, 여기서는 표준 Spring Security 설정을 중심으로 설명합니다.[14]

YAML

```yaml
spring:
  security:
    oauth2:
      client:
        registration:
          azure:
            provider: azure
            client-id: YOUR_CLIENT_ID_FROM_AZURE
            client-secret: YOUR_CLIENT_SECRET_FROM_AZURE
            client-authentication-method: client_secret_basic
            authorization-grant-type: authorization_code
            redirect-uri: '{baseUrl}/login/oauth2/code/{registrationId}'
            scope:
              - openid
              - profile
              - email
        provider:
          azure:
            issuer-uri: https://login.microsoftonline.com/YOUR_TENANT_ID_FROM_AZURE/v2.0
```

다음 표는 Azure Portal의 설정 값과 application.yml의 속성을 명확하게 매핑하여 설정 오류를 방지하는 데 도움을 줍니다.

Azure Portal 설정	**application.yml** 속성	설명
애플리케이션(클라이언트) ID	spring.security.oauth2.client.registration.azure.client-id	Azure에 등록된 애플리케이션의 고유 식별자입니다.
디렉터리(테넌트) ID	spring.security.oauth2.client.provider.azure.issuer-uri의 일부	OIDC 제공자의 발급자 URI를 구성하는 데 사용되는 테넌트 식별자입니다.
클라이언트 암호 값	spring.security.oauth2.client.registration.azure.client-secret	클라이언트 인증에 사용되는 비밀 값입니다.

페더레이션 링크 영속화

Azure AD로부터 받은 토큰을 데이터베이스에 저장하기 위해 JdbcOAuth2AuthorizedClientService 빈을 등록해야 합니다. 이 서비스는 사용자가 인증에 성공했을 때 Azure AD가 발급한 토큰을 oauth2_authorized_client 테이블에 저장하는 역할을 합니다. 이를 통해 사용자의 세션이 유지되고, 필요 시 Refresh Token을 사용하여 새로운 Access Token을 발급받을 수 있습니다.[8]

Java

```
@Bean
public OAuth2AuthorizedClientService authorizedClientService(
    JdbcTemplate jdbcTemplate,
    ClientRegistrationRepository clientRegistrationRepository) {
  return new JdbcOAuth2AuthorizedClientService(jdbcTemplate,
clientRegistrationRepository);
}
```

이 빈을 등록함으로써 ID 허브는 Azure AD와의 연동 관계를 영속적으로 관리할 수 있게 되며, 안정적인 SSO 환경의 기반을 마련합니다.

섹션 **1.3:** 핵심 로직**:** 인증 성공 시 커스텀 **JWT** 발급

표준 OIDC 로그인 흐름을 가로채 커스텀 로직을 실행하고, 강화된 JWT를 발급하는 과정입니다.

인터셉션 포인트**: AuthenticationSuccessHandler**

OIDC 로그인이 성공적으로 완료된 직후, 커스텀 로직을 실행하기 위한 가장 이상적인 지점은 AuthenticationSuccessHandler를 구현하는 것입니다. 이 핸들러는 인증 성공 이벤트가

발생했을 때 호출되어, 리디렉션 또는 추가적인 처리를 수행할 기회를 제공합니다.[21]

CustomAuthenticationSuccessHandler 구현

AuthenticationSuccessHandler 인터페이스를 구현하는 클래스를 생성합니다. onAuthenticationSuccess 메서드 내부에서 다음 로직을 수행합니다.

1. **OidcUser** 정보 추출**:** Authentication 객체의 getPrincipal() 메서드를 통해 인증된 사용자 정보를 가져온 후, 이를 OidcUser 타입으로 캐스팅합니다. 이를 통해 Azure AD가 ID 토큰에 담아 보낸 클레임(이름, 이메일 등)에 접근할 수 있습니다.[15]
2. 사용자 식별 및 **DB** 조회**:** OidcUser에서 email과 같은 고유 식별자를 추출합니다. 이 식별자를 키로 사용하여 내부 사용자 데이터베이스(예: UserRepository를 통해)를 조회하고, 해당 사용자의 역할 목록을 가져옵니다.[24]
3. 로직 분리**:** 실제 JWT 생성 로직은 AuthenticationSuccessHandler에서 직접 처리하기보다, Spring Authorization Server의 공식 확장 지점인 OAuth2TokenCustomizer에 위임하는 것이 좋습니다. 핸들러는 DB에서 조회한 역할 정보를 SecurityContext나 다른 임시 저장소에 저장하여 OAuth2TokenCustomizer가 접근할 수 있도록 하는 역할을 수행할 수 있습니다.

토큰 강화**: OAuth2TokenCustomizer**

Spring Authorization Server에서 JWT에 커스텀 클레임을 추가하는 가장 현대적이고 권장되는 방법은 OAuth2TokenCustomizer<JwtEncodingContext>를 사용하는 것입니다.[26] 이 인터페이스의 빈을 등록하면 토큰 생성 파이프라인에 자동으로 통합됩니다.

Java

```
@Bean
public OAuth2TokenCustomizer<JwtEncodingContext> tokenCustomizer(UserRepository
userRepository) {
    return context -> {
        // Access Token에만 클레임을 추가하도록 제한
        if (OAuth2TokenType.ACCESS_TOKEN.equals(context.getTokenType())) {
            Authentication principal = context.getPrincipal();
```

```
        if (principal instanceof OAuth2AuthenticationToken) {
            OidcUser oidcUser = (OidcUser) principal.getPrincipal();
            String email = oidcUser.getEmail();

            // DB에서 사용자 및 역할 정보 조회
            User user = userRepository.findByEmail(email)
                .orElseThrow(() -> new UsernameNotFoundException("User not found"));

            Set<String> roles = user.getRoles().stream()
                .map(Role::getName)
                .collect(Collectors.toSet());

            // JWT 클레임에 추가
            context.getClaims().claim("roles", roles);
            // Azure AD의 클레임도 필요한 만큼 추가
            context.getClaims().claim("given_name", oidcUser.getGivenName());
            context.getClaims().claim("family_name", oidcUser.getFamilyName());
        }
    }
  };
}
```

이 방식을 통해 생성되는 최종 커스텀 JWT의 클레임 구조는 다음과 같이 명확하게 정의되어야 하며, 이는 모든 다운스트림 서비스와의 계약(contract) 역할을 합니다.

클레임 이름	소스	설명
iss	커스텀 Auth Server	토큰 발급자를 나타냅니다. (예: http://auth-server:9000)
sub	Azure AD email 클레임	토큰의 주체(subject)로, 사용자의 고유 이메일 주소를 사용합니다.
aud	클라이언트 등록 정보	토큰의 대상(audience)으로, 이 토큰을 사용할 리소스 서버를 지정합니다.
exp	커스텀 Auth Server	토큰의 만료 시간을 나타냅니다.

name	Azure AD name 클레임	사용자의 전체 이름입니다.
email	Azure AD email 클레임	사용자의 이메일 주소입니다.
roles	내부 사용자 DB	사용자의 권한을 나타내는 문자열 배열입니다. (예: ``)

SecurityFilterChain에 통합

마지막으로, 생성한 컴포넌트들을 SecurityFilterChain에 통합하여 전체 흐름을 완성합니다.

Java

```java
@Bean
@Order(2)
public SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http,
                                      AuthenticationSuccessHandler customAuthenticationSuccessHandler)
throws Exception {
    http
    .authorizeHttpRequests(authorize ->
            authorize.anyRequest().authenticated()
        )
        // OIDC 로그인을 활성화하고 커스텀 성공 핸들러를 연결합니다.
    .oauth2Login(oauth2Login ->
            oauth2Login.successHandler(customAuthenticationSuccessHandler)
        )
        // 폼 기반 로그인도 함께 제공할 수 있습니다.
    .formLogin(Customizer.withDefaults());

    return http.build();
}
```

이 설정에서 .oauth2Login().successHandler() 체이닝을 통해, Azure AD를 통한 OIDC 인증이 성공하면 지정된 customAuthenticationSuccessHandler가 호출됩니다. 그 후 Spring

Authorization Server의 나머지 설정(registeredClientRepository, jwkSource 등)이 OAuth2TokenCustomizer에 의해 강화된 새로운 JWT를 서명하고 클라이언트에게 최종적으로 발급하게 됩니다.[23]

섹션 **1.4:** 사용자 역할 관리를 위한 **JPA** 엔티티 모델링

사용자와 역할(Role) 간의 다대다(Many-to-Many) 관계를 JPA를 사용하여 모델링합니다. 여기서는 User와 Role 두 개의 엔티티와 이들을 연결하는 조인 테이블 user_roles를 사용합니다.[28]

Role 엔티티

먼저 역할 정보를 담을 Role 엔티티를 정의합니다.

Java

```
// Role.java
@Entity
@Table(name = "roles")
public class Role {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(length = 20)
    private String name;

    // Getters and Setters
}
```

User 엔티티

다음으로 사용자 정보를 담을 User 엔티티를 정의합니다. @ManyToMany 어노테이션을 사용하여 Role과의 관계를 설정하고, @JoinTable을 통해 중간 테이블의 정보를 명시합니다.[31]

Java

```java
// User.java
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true, length = 45)
    private String email;

    @Column(nullable = false, length = 64)
    private String password;

    @Column(name = "full_name", length = 45)
    private String fullName;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(
            name = "user_roles",
            joinColumns = @JoinColumn(name = "user_id"),
            inverseJoinColumns = @JoinColumn(name = "role_id")
    )
    private Set<Role> roles = new HashSet<>();

    // Getters and Setters
}
```

- **@ManyToMany**: User와 Role이 다대다 관계임을 나타냅니다. fetch = FetchType.EAGER는 사용자를 조회할 때 연관된 역할 정보도 즉시 함께 가져오도록 설정합니다.
- **@JoinTable**:
  - name = "user_roles": 관계를 매핑할 조인 테이블의 이름을 지정합니다.
  - joinColumns: 현재 엔티티(User)를 참조하는 조인 테이블의 외래 키 컬럼(user_id)을

지정합니다.
    - inverseJoinColumns: 반대쪽 엔티티(Role)를 참조하는 조인 테이블의 외래 키 컬럼(role_id)을 지정합니다.

UserRepository 리포지토리

Spring Data JPA를 사용하여 데이터베이스 작업을 처리할 리포지토리 인터페이스를 생성합니다.

Java

```
// UserRepository.java
public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByEmail(String email);
}
```

이 설정을 통해 ID 허브는 Azure AD로부터 인증받은 사용자의 이메일을 사용하여 내부 데이터베이스에서 해당 사용자의 역할 목록을 효율적으로 조회하고, 이를 JWT 클레임에 추가할 수 있습니다.

Part II: 리소스 서버**:** 커스텀 **JWT** 소비 및 시행

ID 허브가 발급한 커스텀 JWT를 검증하고, 토큰에 포함된 역할 정보를 기반으로 접근 제어를 수행하는 리소스 서버를 구축합니다.

섹션 **2.1: JWT** 검증을 위한 리소스 서버 설정

프로젝트 설정

Spring Initializr를 사용하여 Spring Web, Spring Security, Thymeleaf, OAuth2 Resource Server 의존성을 포함하는 새로운 Spring Boot 프로젝트를 생성합니다.[3]

보안 설정

리소스 서버의 보안 설정은 매우 간결합니다. SecurityFilterChain 빈을 생성하고, .oauth2ResourceServer(oauth2 -> oauth2.jwt())를 호출하여 JWT 기반의 리소스 서버로 동작하도록 설정합니다. 이 한 줄의 코드는 JWT 수신, 파싱, 서명 검증, 클레임 유효성 검사 등 모든 필요한 파이프라인을 자동으로 구성합니다.[24]

Java

```
@Configuration
@EnableWebSecurity
public class ResourceServerConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
        .authorizeHttpRequests(authorize -> authorize
            .anyRequest().authenticated()
            )
        .oauth2ResourceServer(oauth2 -> oauth2.jwt());
        return http.build();
    }
}
```

가장 중요한 설정은 application.yml 파일에 있습니다. 리소스 서버가 JWT를 검증하기 위해 신뢰할 발급자(issuer)를 지정해야 합니다.

YAML

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://auth-server:9000
```

여기서 issuer-uri는 Azure AD의 주소가 아닌, 우리가 구축한 커스텀 **Authorization Server**의 주소를 가리켜야 합니다.[32]

이 설정은 아키텍처의 핵심적인 특징인 '신뢰 경계의 이동'을 기술적으로 구현한 것입니다. 일반적인 리소스 서버는 issuer-uri를 Azure AD 테넌트 URL로 설정하여 Azure AD를 직접 신뢰합니다.[35] 하지만 ID 허브 아키텍처에서 리소스 서버는 Azure AD의 존재 자체를 알지 못합니다. 리소스 서버의 유일한 신뢰 대상은 우리의 커스텀 Authorization Server입니다. 이로 인해 리소스 서버의 설정과 로직은 매우 단순해집니다. 향후 ID 허브가 여러 외부 IdP를 지원하게 되더라도 리소스 서버는 오직 하나의 토큰 형식과 하나의 발급자만 신뢰하면 되므로, 변경에 대한 영향을 전혀 받지 않습니다.

섹션 **2.2:** 역할 기반 접근 제어**(RBAC)** 구현

JWT 검증이 완료되면, 토큰 내의 roles 클레임을 사용하여 세분화된 접근 제어를 구현할 수 있습니다.

메서드 보안 활성화

보안 설정 클래스에 @EnableMethodSecurity 어노테이션을 추가하여 메서드 수준의 보안 어노테이션 사용을 활성화합니다.[25]

API 수준의 **RBAC**

@RestController의 각 엔드포인트 메서드에 @PreAuthorize 어노테이션을 사용하여 접근 제어를 적용합니다.

- 스코프 기반 제어**:** @PreAuthorize("hasAuthority('SCOPE_read')")는 클라이언트가 read

스코프를 요청했는지 확인하는 데 사용됩니다.

- 역할 기반 제어: @PreAuthorize("hasRole('ADMIN')")는 인증된 사용자의 JWT에 ADMIN 역할이 포함되어 있는지 확인합니다. Spring Security는 JWT의 roles 클레임에 있는 각 역할을 자동으로 ROLE_ 접두사를 붙인 GrantedAuthority로 변환하므로, 코드에서는 접두사 없이 역할 이름만 사용하면 됩니다.[25]

Java

```
@RestController
public class ApiController {

    @GetMapping("/api/public")
    public String publicEndpoint() {
        return "This is a public endpoint.";
    }

    @GetMapping("/api/user")
    @PreAuthorize("hasRole('USER')")
    public String userEndpoint() {
        return "Hello, User!";
    }

    @GetMapping("/api/admin")
    @PreAuthorize("hasRole('ADMIN')")
    public String adminEndpoint() {
        return "Hello, Admin!";
    }
}
```

UI 수준의 RBAC (Thymeleaf)

Thymeleaf 템플릿 엔진과 Spring Security를 통합하여 사용자의 역할에 따라 UI 요소를 동적으로 렌더링할 수 있습니다. 이를 위해 thymeleaf-extras-springsecurity6 의존성을 추가해야 합니다.

템플릿 파일 내에서 sec: 네임스페이스를 사용하여 보안 관련 기능에 접근할 수 있습니다.[36]

- 인증된 사용자 이름 표시:
  HTML
  ```
  <p>Welcome, <span sec:authentication="name">User</span>!</p>
  ```

- 역할에 따른 조건부 렌더링:
  HTML
  ```
  <div sec:authorize="hasRole('ADMIN')">
    <a href="/admin/dashboard">Admin Dashboard</a>
  </div>
  <div sec:authorize="hasRole('USER')">
    <p>This content is visible to all authenticated users.</p>
  </div>
  ```

이러한 방식을 통해 API와 UI 양쪽에서 일관된 역할 기반 접근 제어를 구현하여 강력하고 유연한 보안 모델을 완성할 수 있습니다.

Part III: 클라이언트**:** 웹 및 데스크톱 애플리케이션 통합

ID 허브와 상호작용하는 Nuxt.js 웹 클라이언트와.NET WinForms 데스크톱 클라이언트를 구현하는 방법을 설명합니다. 각 클라이언트에서 API 요청을 자동화하고 보안을 강화하는 기술에 중점을 둡니다.

섹션 **3.1: Nuxt.js (SSR)** 웹 클라이언트 구현

서버 사이드 렌더링**(SSR)**과 **OIDC** 흐름

Nuxt.js와 같은 서버 사이드 렌더링(SSR) 프레임워크에서 OIDC 인증을 구현할 때는 보안을 위해 클라이언트 측(브라우저)이 아닌 서버 측에서 토큰 교환을 처리하는 것이 중요합니다. 이는 client_secret과 같은 민감한 정보를 노출로부터 보호합니다.

1. 로그인 시작**:** 사용자가 로그인 링크를 클릭하면, Nuxt 앱은 인증 서버의 /oauth2/authorize 엔드포인트로 리디렉션하는 서버 라우트(예: /api/auth/login)를 호출합니다.
2. 콜백 처리**:** 인증 성공 후, 인증 서버는 Nuxt 앱의 서버 콜백 라우트(예: /api/auth/callback)로

authorization_code와 함께 사용자를 리디렉션합니다.

3. 서버 측 토큰 교환**:** 콜백 라우트는 수신한 code를 사용하여 인증 서버의 /oauth2/token 엔드포인트에 백그라운드에서 POST 요청을 보냅니다. 이 요청에는 code, client_id, redirect_uri, 그리고 PKCE를 위한 code_verifier가 포함됩니다.
4. 세션 생성**:** 인증 서버로부터 JWT(Access Token)를 성공적으로 수신하면, Nuxt 서버는 이 토큰을 암호화된 httpOnly 쿠키에 저장하여 사용자 세션을 생성합니다. 이 방식은 XSS 공격으로부터 토큰을 보호하는 가장 안전한 방법 중 하나입니다.
5. 인증 완료**:** 사용자는 애플리케이션의 홈 페이지로 리디렉션되며, 이제 인증된 상태가 됩니다.

API 호출 자동화를 위한 **Nuxt 3** 플러그인

Nuxt 3에서는 플러그인을 사용하여 $fetch의 동작을 전역적으로 확장할 수 있습니다. 이는 과거 Axios 인터셉터와 유사한 역할을 수행합니다. API 호출 시 자동으로 인증 헤더를 추가하고 오류를 처리하는 플러그인을 작성합니다.

plugins/api.ts

TypeScript

```
import { ofetch } from 'ofetch'

export default defineNuxtPlugin((_nuxtApp) => {
 const config = useRuntimeConfig()

 globalThis.$api = ofetch.create({
  baseURL: config.public.apiBase,

  // 요청 인터셉터 (Request Interceptor)
  onRequest({ options }) {
   // 이 로직은 서버 사이드 렌더링(SSR) 중에 Nuxt 서버가 API 서버로 요청을 보낼 때 실행됩니다.
   // httpOnly 쿠키는 서버 측에서만 접근 가능합니다.
   if (process.server) {
    const event = useRequestEvent()
    // useRequestEvent()는 서버 컨텍스트에서만 사용 가능합니다.
    if (event) {
     const token = getCookie(event, 'auth_token')
```

```
        if (token) {
          // 헤더 객체가 없으면 새로 생성합니다.
          const headers = (options.headers ||= {}) as Record<string, string>
          headers.Authorization = `Bearer ${token}`
        }
      }
    }
  },

  // 응답 에러 인터셉터 (Response Error Interceptor)
  onResponseError({ response }) {
    // 401 Unauthorized 에러 발생 시 로그인 페이지로 리디렉션합니다.
    if (response.status === 401) {
      // 클라이언트 측에서만 리디렉션 수행
      if (process.client) {
        // 토큰이 만료되었으므로 로그인 프로세스를 다시 시작합니다.
        window.location.href = '/api/auth/login'
      }
    }
  }
 })
})
```

이 플러그인은 $api라는 전역 헬퍼를 생성합니다. onRequest 훅은 서버 측에서 실행될 때 auth_token 쿠키를 읽어 모든 나가는 API 요청에 Authorization 헤더를 자동으로 추가합니다. onResponseError 훅은 API가 401 응답을 반환하면(토큰 만료 등) 사용자를 로그인 프로세스로 다시 안내합니다.

섹션 **3.2:.NET WinForms** 데스크톱 클라이언트 및 보안 강화

네이티브 데스크톱 애플리케이션을 위한 OIDC 흐름을 구현하고, 특히 리버스 엔지니어링으로부터 JWT를 보호하는 데 중점을 둡니다.

네이티브 앱 **OIDC** 흐름 및 **DelegatingHandler**

.NET 환경에서는 IdentityModel.OidcClient 라이브러리를 사용하여 네이티브 애플리케이션의 OIDC 흐름을 간편하게 구현할 수 있습니다.[37] 웹 앱과 달리 고정된 리디렉션 URI를 호스팅할 수

없으므로, 로컬 루프백 주소(예: http://127.0.0.1:7890)를 리디렉션 URI로 사용합니다.

API 호출을 자동화하고 중앙에서 관리하기 위해 DelegatingHandler를 사용합니다. 이는 ASP.NET Core의 미들웨어 파이프라인과 유사하게 HttpClient의 요청/응답 흐름에 개입할 수 있는 강력한 메커니즘입니다.

AuthenticationHandler.cs

C#

```
using System.Net;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Threading;
using System.Threading.Tasks;

public class AuthenticationHandler : DelegatingHandler
{
    public AuthenticationHandler()
    {
        // 실제 HTTP 요청을 처리할 내부 핸들러를 설정합니다.
        InnerHandler = new HttpClientHandler();
    }

    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        // DPAPI를 통해 안전하게 저장된 토큰을 로드합니다.
        var token = SecureTokenStorage.LoadToken();
        if (!string.IsNullOrEmpty(token))
        {
            request.Headers.Authorization = new AuthenticationHeaderValue("Bearer", token);
        }

        // 원래 요청을 보냅니다.
        var response = await base.SendAsync(request, cancellationToken);

        // 응답이 401 Unauthorized인 경우 처리합니다.
        if (response.StatusCode == HttpStatusCode.Unauthorized)
        {
```

```
            // 이 아키텍처에서는 클라이언트가 Refresh Token을 직접 관리하지 않으므로,
            // 401은 Access Token이 만료되었음을 의미하며, 사용자는 다시 로그인해야 합니다.
            // 만약 Refresh Token 흐름이 구현된다면, 여기서 토큰 갱신 로직을 호출하고
            // 원래 요청을 재시도할 수 있습니다.

            // 현재로서는 저장된 만료된 토큰을 삭제하여 다음 요청 시 불필요한 시도를 막습니다.
            SecureTokenStorage.DeleteToken();
        }

        return response;
    }
}
```

이 AuthenticationHandler는 모든 HttpClient 요청을 가로채 SecureTokenStorage에서 토큰을 로드하여 Authorization 헤더를 자동으로 추가합니다. 또한, 응답이 401일 경우 저장된 토큰을 삭제하여 후속 문제를 방지합니다.

심층 분석: DPAPI를 이용한 JWT 보안

Windows DPAPI(Data Protection API)는 이러한 위협에 대응하기 위한 효과적인 솔루션입니다. DPAPI는 Windows 운영체제가 제공하는 암호화 서비스로, 사용자의 Windows 로그인 자격 증명이나 컴퓨터의 자격 증명에 암호화 키를 연동합니다. 이를 통해 개발자는 복잡한 키 관리 문제 없이 데이터를 안전하게 보호할 수 있습니다.[38]

.NET에서는 System.Security.Cryptography.ProtectedData 클래스를 통해 DPAPI에 접근할 수 있습니다. 다음은 JWT를 안전하게 저장하고 불러오기 위한 C# 헬퍼 클래스 예제입니다.[40]

C#

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;

public static class SecureTokenStorage
{
    // 추가적인 보안 강화를 위한 엔트로피 (애플리케이션 고유 값으로 설정)
```

```
    private static readonly byte s_entropy = Encoding.UTF8.GetBytes("MySuperSecretEntropy");
    private static readonly string s_tokenFilePath = Path.Combine(
        Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData),
        "MyAppName",
        "token.dat");

    public static void SaveToken(string token)
    {
        byte tokenBytes = Encoding.UTF8.GetBytes(token);
        byte encryptedToken = ProtectedData.Protect(tokenBytes, s_entropy,
DataProtectionScope.CurrentUser);
        Directory.CreateDirectory(Path.GetDirectoryName(s_tokenFilePath));
        File.WriteAllBytes(s_tokenFilePath, encryptedToken);
    }

    public static string LoadToken()
    {
        if (!File.Exists(s_tokenFilePath)) return null;
        byte encryptedToken = File.ReadAllBytes(s_tokenFilePath);
        try
        {
            byte tokenBytes = ProtectedData.Unprotect(encryptedToken, s_entropy,
DataProtectionScope.CurrentUser);
            return Encoding.UTF8.GetString(tokenBytes);
        }
        catch (CryptographicException)
        {
            File.Delete(s_tokenFilePath);
            return null;
        }
    }

    public static void DeleteToken()
    {
        if (File.Exists(s_tokenFilePath))
        {
            File.Delete(s_tokenFilePath);
        }
    }
}
```

이 클래스의 SaveToken 메서드는 JWT 문자열을 받아 ProtectedData.Protect를 호출합니다. 이때 DataProtectionScope.CurrentUser 옵션을 사용하여, 암호화된 데이터는 오직 현재

로그인한 Windows 사용자만이 복호화할 수 있도록 보장합니다.[41] 암호화된 바이트 배열은 파일에 저장됩니다.

Part IV: 전체 구현 프로젝트 코드

이 섹션에서는 앞에서 설명한 아키텍처를 기반으로 실제 동작하는 전체 프로젝트 코드를 제공합니다. 각 프로젝트는 독립적인 디렉토리로 구성되어 있습니다.

섹션 **4.1:** 인증 서버 **(auth-server)**

(이전 응답과 동일)

섹션 **4.2:** 리소스 서버 **(resource-server)**

(이전 응답과 동일)

섹션 **4.3: Nuxt.js** 클라이언트 **(nuxt-client)**

OIDC PKCE 흐름을 서버 사이드에서 처리하는 Nuxt 3 SSR 클라이언트입니다.

package.json

JSON

```
{
  "private": true,
  "scripts": {
```

```
    "build": "nuxt build",
    "dev": "nuxt dev",
    "generate": "nuxt generate",
    "preview": "nuxt preview"
  },
  "dependencies": {
    "nuxt": "^3.10.0",
    "ofetch": "^1.3.3"
  }
}
```

nuxt.config.ts

```
    TypeScript
```

```
export default defineNuxtConfig({
  runtimeConfig: {
    oauth: {
      authServerUrl: 'http://localhost:9000',
      clientId: 'nuxt-client',
      clientSecret: 'secret', // SSR에서는 시크릿 사용 가능하지만, public client이므로 불필요
      tokenEndpoint: 'http://localhost:9000/oauth2/token',
      authorizationEndpoint: 'http://localhost:9000/oauth2/authorize'
    },
    public: {
      apiBase: 'http://localhost:8082/api'
    }
  }
})
```

server/api/auth/login.get.ts

```
    TypeScript
```

```
import { nanoid } from 'nanoid'

export default defineEventHandler((event) => {
```

```
  const config = useRuntimeConfig(event).oauth

  const codeVerifier = nanoid(32)
  // TODO: code_verifier를 세션에 저장해야 함

  const params = new URLSearchParams()
  params.append('response_type', 'code')
  params.append('client_id', config.clientId)
  params.append('scope', 'openid profile')
  params.append('redirect_uri', 'http://localhost:3000/api/auth/callback')
  // params.append('code_challenge', createCodeChallenge(codeVerifier))
  // params.append('code_challenge_method', 'S256')

  return sendRedirect(event, `${config.authorizationEndpoint}?${params.toString()}`)
})
```

server/api/auth/callback.get.ts

```
    TypeScript
```

```
import { ofetch } from 'ofetch'

export default defineEventHandler(async (event) => {
  const config = useRuntimeConfig(event).oauth
  const { code } = getQuery(event)

  // TODO: 세션에서 code_verifier를 가져와야 함

  const body = new URLSearchParams()
  body.append('grant_type', 'authorization_code')
  body.append('code', code as string)
  body.append('redirect_uri', 'http://localhost:3000/api/auth/callback')
  body.append('client_id', config.clientId)
  // body.append('code_verifier', storedCodeVerifier)

  try {
    const tokens = await ofetch(config.tokenEndpoint, {
      method: 'POST',
      headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
      body: body
```

```
  })

  setCookie(event, 'auth_token', tokens.access_token, {
    httpOnly: true,
    secure: process.env.NODE_ENV === 'production',
    sameSite: 'lax',
    maxAge: tokens.expires_in
  })

  return sendRedirect(event, '/')
 } catch (error) {
  console.error("Token exchange failed:", error)
  return sendRedirect(event, '/login-failed')
 }
})
```

섹션 **4.4: WinForms** 클라이언트 **(winforms-client)**

.NET WinForms 애플리케이션으로, DelegatingHandler를 사용하여 API 호출을 자동화하고 DPAPI로 토큰을 안전하게 저장합니다.

ApiClient.cs

C#

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

public class ApiClient
{
  private readonly HttpClient _httpClient;

  public ApiClient()
  {
    // AuthenticationHandler를 HttpClient 파이프라인에 연결합니다.
    _httpClient = new HttpClient(new AuthenticationHandler())
```

```
        {
            BaseAddress = new Uri("http://localhost:8082/api/")
        };
    }

    public async Task<string> GetUserResourceAsync()
    {
        try
        {
            HttpResponseMessage response = await _httpClient.GetAsync("user");
            response.EnsureSuccessStatusCode();
            return await response.Content.ReadAsStringAsync();
        }
        catch (HttpRequestException ex) when (ex.StatusCode ==
System.Net.HttpStatusCode.Unauthorized)
        {
            return "Access Denied. Please log in again.";
        }
        catch (Exception ex)
        {
            return $"An error occurred: {ex.Message}";
        }
    }
}
```

MainForm.cs (수정된 부분)

C#

```
public partial class MainForm : Form
{
    private readonly OidcClientService _oidcClientService;
    private readonly ApiClient _apiClient;

    public MainForm()
    {
        InitializeComponent();
        _oidcClientService = new OidcClientService();
        _apiClient = new ApiClient(); // ApiClient 인스턴스 생성
    }
```

```
    //... btnLogin_Click 이벤트 핸들러는 동일...

    private async void btnCallApi_Click(object sender, EventArgs e)
    {
        // ApiClient를 통해 API 호출, 헤더 설정이 자동으로 처리됨
        var response = await _apiClient.GetUserResourceAsync();
        txtLog.Text = $"API Response: {response}";
    }
}
```

결론: 고급 고려사항 및 모범 사례

(이전 응답과 동일)

참고 자료

1. Spring Security - Authentication Providers - GeeksforGeeks, 10월 18, 2025에 액세스, https://www.geeksforgeeks.org/java/spring-security-authentication-providers/
2. Implementing Secure JWT Authentication and Role-Based Authorization in Spring Boot, 10월 18, 2025에 액세스, https://medium.com/@yesithaathukorala/implementing-secure-jwt-authentication-and-role-based-authorization-in-spring-boot-ae91696c6a79
3. Add sign-in with Azure Active Directory B2C to a Spring Web App - Microsoft Learn, 10월 18, 2025에 액세스, https://learn.microsoft.com/en-us/azure/developer/java/spring-framework/configure-spring-boot-starter-java-app-with-azure-active-directory-b2c-oidc
4. Spring Authorization Server, 10월 18, 2025에 액세스, https://spring.io/projects/spring-authorization-server/
5. Build OAuth2.0 SpringBoot, 10월 18, 2025에 액세스, https://guide.ncloud-docs.com/docs/en/b2bpls-oauth2springboot
6. Spring Authorization Server : 0–90 | by Afeef Razick | Medium, 10월 18, 2025에 액세스, https://medium.com/@afeefrazickamir/spring-authorization-server-0-90-03d996d5c5a7
7. OAuth2 in Spring Security: Understanding the Client, Authorization Server, and Resource Server | by Jefster | Medium, 10월 18, 2025에 액세스, https://medium.com/@dev.jefster/oauth2-in-spring-security-understanding-the-client-authorization-server-and-resource-server-e90c14630b20

8. Spring Security Persistent OAuth2 Client - DEV Community, 10월 18, 2025에 액세스, https://dev.to/relive27/spring-security-persistent-oauth2-client-1gcc
9. Spring Boot with H2 Database - GeeksforGeeks, 10월 18, 2025에 액세스, https://www.geeksforgeeks.org/springboot/spring-boot-with-h2-database/
10. How can I provide different database configurations with Spring Boot? - Stack Overflow, 10월 18, 2025에 액세스, https://stackoverflow.com/questions/28007686/how-can-i-provide-different-database-configurations-with-spring-boot
11. Using H2 and Oracle with Spring Boot - Spring Framework Guru, 10월 18, 2025에 액세스, https://springframework.guru/using-h2-and-oracle-with-spring-boot/
12. Access Token Refresh Flows in Spring Boot Security - Medium, 10월 18, 2025에 액세스, https://medium.com/@AlexanderObregon/access-token-refresh-flows-in-spring-boot-security-d3cfe55b2182
13. Advanced OAuth2: Refresh Tokens and Token Expiration Strategies - Igor Venturelli, 10월 18, 2025에 액세스, https://igventurelli.io/advanced-oauth2-refresh-tokens-and-token-expiration-strategies/
14. Spring Cloud Azure Spring Security support - Java on Azure | Microsoft Learn, 10월 18, 2025에 액세스, https://learn.microsoft.com/en-us/azure/developer/java/spring-framework/spring-security-support
15. Secure Java Spring Boot apps using roles and role claims - Azure - Microsoft Learn, 10월 18, 2025에 액세스, https://learn.microsoft.com/en-us/azure/developer/java/identity/enable-spring-boot-webapp-authorization-role-entra-id
16. Spring Boot Starter for Microsoft Entra developer's guide - Java on Azure, 10월 18, 2025에 액세스, https://learn.microsoft.com/en-us/azure/developer/java/spring-framework/spring-boot-starter-for-entra-developer-guide
17. Getting Started | Spring Boot and OAuth2, 10월 18, 2025에 액세스, https://spring.io/guides/tutorials/spring-boot-oauth2/
18. Spring Cloud Azure, 10월 18, 2025에 액세스, https://spring.io/projects/spring-cloud-azure/
19. JdbcOAuth2AuthorizedClientSer, 10월 18, 2025에 액세스, https://docs.spring.io/spring-security/site/docs/5.4.2/api/index.html?org/springframework/security/oauth2/client/JdbcOAuth2AuthorizedClientService.html
20. JdbcOAuth2AuthorizedClientSer, 10월 18, 2025에 액세스, https://github.com/spring-projects/spring-security/issues/8425
21. Spring Custom AuthenticationSuccessHandler Example ..., 10월 18, 2025에 액세스, https://javapointers.com/spring/spring-security/spring-custom-authenticationsuccesshandler-example-2/
22. Spring Security — Demonstrating Custom Authentication Success Handler | by Syed Hasan, 10월 18, 2025에 액세스,

https://mainul35.medium.com/spring-security-demonstrating-custom-authentication-success-handler-3b6fcb572a53
23. Spring Security – OAuth2 Login - Baeldung, 10월 18, 2025에 액세스, https://www.baeldung.com/spring-security-5-oauth2-login
24. Spring Boot Token based Authentication with Spring Security & JWT ..., 10월 18, 2025에 액세스, https://www.bezkoder.com/spring-boot-jwt-authentication/
25. Build a Role-based Access Control in a Spring Boot 3 API - Teco Tutorials, 10월 18, 2025에 액세스, https://blog.tericcabrel.com/role-base-access-control-spring-boot/
26. Spring OAuth Server: Token Claim Customization - DZone, 10월 18, 2025에 액세스, https://dzone.com/articles/spring-oauth-server-token-claim-customization
27. Issuing a Custom JWT After Authentication - oauth 2.0 - Stack Overflow, 10월 18, 2025에 액세스, https://stackoverflow.com/questions/74123667/issuing-a-custom-jwt-after-authentication
28. Spring Boot Many To Many Relationship | by HK - Medium, 10월 18, 2025에 액세스, https://medium.com/@hk09/spring-boot-many-to-many-relationship-ac0ffd9b545d
29. JPA - Many-To-Many Mapping - GeeksforGeeks, 10월 18, 2025에 액세스, https://www.geeksforgeeks.org/advance-java/jpa-many-to-many-mapping/
30. Many-To-Many Relationship in JPA - Baeldung, 10월 18, 2025에 액세스, https://www.baeldung.com/jpa-many-to-many
31. JPA User, Role, manyToMany hibernate relationship - Stack Overflow, 10월 18, 2025에 액세스, https://stackoverflow.com/questions/43600797/jpa-user-role-manytomany-hibernate-relationship
32. OAuth 2.0 Resource Server JWT :: Spring Security, 10월 18, 2025에 액세스, https://docs.spring.io/spring-security/reference/reactive/oauth2/resource-server/jwt.html
33. What is `issuer-uri` in conext of Spring Security? (rant about Spring Security documentation), 10월 18, 2025에 액세스, https://www.reddit.com/r/SpringBoot/comments/1k8934h/what_is_issueruri_in_conext_of_spring_security/
34. OAuth 2.0 Resource Server With Spring Security - Baeldung, 10월 18, 2025에 액세스, https://www.baeldung.com/spring-security-oauth-resource-server
35. How to integrate spring security with Azure OAuth2.0 endpoint ? - Microsoft Q&A, 10월 18, 2025에 액세스, https://learn.microsoft.com/en-us/answers/questions/1803199/how-to-integrate-spring-security-with-azure-oauth2
36. Thymeleaf + Spring Security integration basics, 10월 18, 2025에 액세스, https://www.thymeleaf.org/doc/articles/springsecurity.html
37. auth0/auth0-oidc-client-net: OIDC Client for .NET Desktop and Mobile applications - GitHub, 10월 18, 2025에 액세스, https://github.com/auth0/auth0-oidc-client-net

38. How to: Use Data Protection - .NET | Microsoft Learn, 10월 18, 2025에 액세스, https://learn.microsoft.com/en-us/dotnet/standard/security/how-to-use-data-protection
39. How to encrypt and decrypt data using DPAPI in C# or VB.NET - freesilo Blog, 10월 18, 2025에 액세스, https://freesilo.com/?p=452
40. C# DPAPI Example Usage · GitHub, 10월 18, 2025에 액세스, https://gist.github.com/clod81/f608f63f93cbc379d31090fe9d1298e5
41. ProtectedData Class (System.Security.Cryptography) | Microsoft Learn, 10월 18, 2025에 액세스, https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.protecteddata?view=windowsdesktop-9.0
42. Using the DPAPI through ProtectedData Class in .Net Framework 2.0 - C# Corner, 10월 18, 2025에 액세스, https://www.c-sharpcorner.com/article/using-the-dpapi-through-protecteddata-class-in-net-framewor/