**Overview of Authorization and Authentication in Spring Boot**

**Authentication**: Verifies "who you are" (e.g., username/password).  
**Authorization**: Determines "what you are allowed to do" (e.g., roles/permissions).

Spring Security uses tokens (like JWT) to secure communication between clients and servers.

**Key Classes and Their Purpose (Spring Security Authorization Server)**

1. **RegisteredClient**  
   Stores details of clients (applications) allowed to access your server, including client ID, secret, and permissions.
2. **RegisteredClientRepository**  
   Manages storage of RegisteredClient instances, fetching client details during authentication.
3. **AuthorizationServerSettings**  
   Configures the URLs for OAuth2 endpoints (like /oauth/token for token generation).
4. **OAuth2TokenCustomizer**  
   Adds custom claims to the token, such as user-specific data or extra metadata.
5. **JwtEncoder**  
   Encodes data (claims like user info, roles) into a signed JWT during token creation.
6. **JwtDecoder**  
   Reads the token, verifies its signature, and extracts data (claims) when a client sends it back.
7. **AuthorizationServerConfigurer**  
   Central configuration class for setting up OAuth2 authorization flows, endpoints, and security rules.
8. **AuthorizationGrantType**  
   Defines how a token is issued (e.g., authorization\_code, refresh\_token, or client\_credentials).
9. **OAuth2AccessToken**  
   Represents the token issued to the client, used to access secured resources.
10. **OAuth2RefreshToken**  
    Allows clients to get a new access token without re-authenticating.

**Token Lifecycle Explained**

1. **Creating a Token**  
   The server verifies the user's credentials, adds claims (like roles), and signs them with a private key using JwtEncoder.
2. **Reading a Token**  
   When a client sends the token back (e.g., in a request header), the server uses JwtDecoder to verify its signature and extract user data.
3. **Spring Security + Tokens**
   * Filters intercept requests to validate the token.
   * SecurityContextHolder stores authenticated user details.
   * Custom claims from the token determine access to specific endpoints/resources.

This flow secures your application by authenticating the user once and validating tokens for future requests.

4o

**1. RegisteredClient**

This class represents the details of an OAuth2 client (like an application interacting with your authorization server). It includes the client ID, secret, redirect URIs, grant types, etc.

**Example:**

java

Copy code

RegisteredClient registeredClient = RegisteredClient.withId(UUID.randomUUID().toString())

.clientId("client-id")

.clientSecret("{noop}client-secret") // NoOpPasswordEncoder for simplicity

.redirectUri("http://localhost:8080/login/oauth2/code/my-client")

.authorizationGrantType(AuthorizationGrantType.AUTHORIZATION\_CODE)

.authorizationGrantType(AuthorizationGrantType.REFRESH\_TOKEN)

.scope("read")

.scope("write")

.build();

**2. RegisteredClientRepository**

This interface manages the storage and retrieval of RegisteredClient instances. You can implement it using an in-memory store or a database.

**Example (In-memory):**

java

Copy code

@Bean

public RegisteredClientRepository registeredClientRepository() {

return new InMemoryRegisteredClientRepository(registeredClient);

}

**3. AuthorizationServerSettings**

Configures key URLs like token endpoints (/oauth/token) and authorization endpoints (/oauth/authorize).

**Example:**

java

Copy code

@Bean

public AuthorizationServerSettings authorizationServerSettings() {

return AuthorizationServerSettings.builder()

.tokenEndpoint("/oauth2/token")

.authorizationEndpoint("/oauth2/authorize")

.build();

}

**4. OAuth2TokenCustomizer**

Allows you to add custom claims (extra data) to the token, like user-specific details or application-specific information.

**Example:**

java

Copy code

@Bean

public OAuth2TokenCustomizer<JwtEncodingContext> tokenCustomizer() {

return context -> {

if (context.getTokenType().equals(OAuth2TokenType.ACCESS\_TOKEN)) {

context.getClaims().claim("custom-claim", "custom-value");

}

};

}

**5. JwtEncoder**

Responsible for creating (signing) JWT tokens. It uses a private key to sign the token securely.

**Example:**

java

Copy code

@Bean

public JwtEncoder jwtEncoder() {

JWKSource<SecurityContext> jwkSource = new ImmutableJWKSet<>(new JWKSet(myRsaKey()));

return new NimbusJwtEncoder(jwkSource);

}

**6. JwtDecoder**

Reads and verifies the token signature to extract claims. It ensures the token is valid and trusted.

**Example:**

java

Copy code

@Bean

public JwtDecoder jwtDecoder() {

return NimbusJwtDecoder.withPublicKey(myPublicKey()).build();

}

**7. AuthorizationServerConfigurer**

Central configuration class for enabling OAuth2 flows and token issuance.

**Example:**

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

OAuth2AuthorizationServerConfiguration.applyDefaultSecurity(http);

return http.formLogin().and().build();

}

**8. AuthorizationGrantType**

Defines how tokens are issued. The most common grant types are AUTHORIZATION\_CODE and REFRESH\_TOKEN.

**Example:**

RegisteredClient registeredClient = RegisteredClient.withId(UUID.randomUUID().toString())

.authorizationGrantType(AuthorizationGrantType.AUTHORIZATION\_CODE)

.authorizationGrantType(AuthorizationGrantType.REFRESH\_TOKEN)

.build();

**9. OAuth2AccessToken**

Represents an issued token. This is sent to the client and used to access secured resources.

**Example (as part of the token response):**

java

Copy code

OAuth2AccessToken token = new OAuth2AccessToken(OAuth2AccessToken.TokenType.BEARER,

"token-value", Instant.now(), Instant.now().plusSeconds(3600));

**10. OAuth2RefreshToken**

Allows clients to refresh the access token without requiring re-authentication.

**Example:**

java

Copy code

OAuth2RefreshToken refreshToken = new OAuth2RefreshToken("refresh-token-value", Instant.now());

**Complete Token Lifecycle**

1. **Creating a Token**
   * When a client sends a valid request, the authorization server validates credentials, generates claims, and uses JwtEncoder to create a signed token.

**Example:**

java

Copy code

JwtClaimsSet claims = JwtClaimsSet.builder()

.issuer("http://auth-server.com")

.subject("user")

.claim("roles", List.of("USER", "ADMIN"))

.expiresAt(Instant.now().plusSeconds(3600))

.build();

JwtEncoder encoder = jwtEncoder();

String token = encoder.encode(JwtEncoderParameters.from(claims)).getTokenValue();

1. **Reading a Token**
   * The resource server decodes the token using JwtDecoder, verifies the signature, and extracts claims.

**Example:**

java

Copy code

Jwt jwt = jwtDecoder().decode(token);

String subject = jwt.getSubject();

List<String> roles = jwt.getClaim("roles");

1. **Spring Security + Tokens**
   * Spring Security adds a filter (like BearerTokenAuthenticationFilter) to validate incoming tokens. If valid, it populates SecurityContextHolder with user details.

**Example (Filter Configuration):**

java

Copy code

@Bean

public SecurityFilterChain resourceServerSecurity(HttpSecurity http) throws Exception {

return http.oauth2ResourceServer().jwt().and().build();

}

These classes and configurations work together to handle OAuth2 authentication, issue tokens, validate them, and secure your application endpoints. Let me know if you'd like a specific part expanded further!

2/2

Top of Form

Bottom of Form

/\*

1. RegisteredClient

\*/

import org.springframework.security.oauth2.server.authorization.client.RegisteredClient;

import org.springframework.security.oauth2.server.authorization.client.RegisteredClientRepository;

import org.springframework.security.oauth2.core.ClientAuthenticationMethod;

import org.springframework.security.oauth2.core.AuthorizationGrantType;

RegisteredClient registeredClient = RegisteredClient.withId("client-id")

.clientId("my-client")

.clientSecret("my-secret")

.clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT\_SECRET\_BASIC)

.authorizationGrantType(AuthorizationGrantType.AUTHORIZATION\_CODE)

.authorizationGrantType(AuthorizationGrantType.REFRESH\_TOKEN)

.redirectUri("http://localhost:8080/callback")

.build();

/\*

The RegisteredClient stores client details such as the ID, secret, redirect URI, and grant types.

\*/

/\*

2. RegisteredClientRepository

\*/

import org.springframework.security.oauth2.server.authorization.client.InMemoryRegisteredClientRepository;

RegisteredClientRepository clientRepository = new InMemoryRegisteredClientRepository(registeredClient);

/\*

This manages RegisteredClient instances, enabling client details to be retrieved during authentication.

\*/

/\*

3. AuthorizationServerSettings

\*/

import org.springframework.security.oauth2.server.authorization.config.annotation.web.configuration.OAuth2AuthorizationServerConfiguration;

import org.springframework.security.oauth2.server.authorization.settings.AuthorizationServerSettings;

AuthorizationServerSettings settings = AuthorizationServerSettings.builder()

.tokenEndpoint("/oauth/token")

.build();

/\*

AuthorizationServerSettings configures the endpoints for token operations such as token generation.

\*/

/\*

4. OAuth2TokenCustomizer

\*/

import org.springframework.security.oauth2.server.authorization.token.JwtEncodingContext;

import org.springframework.security.oauth2.server.authorization.token.OAuth2TokenCustomizer;

public class CustomTokenCustomizer implements OAuth2TokenCustomizer<JwtEncodingContext> {

@Override

public void customize(JwtEncodingContext context) {

context.getClaims().claim("custom-claim", "custom-value");

}

}

/\*

CustomTokenCustomizer adds custom claims to tokens such as user-specific metadata.

\*/

/\*

5. JwtEncoder

\*/

import org.springframework.security.oauth2.jwt.Jwt;

import org.springframework.security.oauth2.jwt.JwtEncoder;

import org.springframework.security.oauth2.jwt.NimbusJwtEncoder;

import org.springframework.security.oauth2.jose.jws.SignatureAlgorithm;

import org.springframework.security.oauth2.jose.jws.JwsHeader;

import org.springframework.security.oauth2.jwt.JwtClaimsSet;

import java.time.Instant;

JwtClaimsSet claims = JwtClaimsSet.builder()

.issuer("http://localhost")

.subject("user-id")

.issuedAt(Instant.now())

.expiresAt(Instant.now().plusSeconds(3600))

.build();

JwtEncoder jwtEncoder = new NimbusJwtEncoder(jwkSource);

Jwt jwt = jwtEncoder.encode(JwsHeader.with(SignatureAlgorithm.RS256).build(), claims);

/\*

JwtEncoder generates a signed token based on claims and headers.

\*/

/\*

6. JwtDecoder

\*/

import org.springframework.security.oauth2.jwt.JwtDecoder;

import org.springframework.security.oauth2.jwt.NimbusJwtDecoder;

JwtDecoder jwtDecoder = NimbusJwtDecoder.withPublicKey(publicKey).build();

Jwt decodedJwt = jwtDecoder.decode("your-jwt-token");

System.out.println(decodedJwt.getClaims());

/\*

JwtDecoder validates the token signature and extracts claims for further use.

\*/

/\*

7. AuthorizationServerConfigurer

\*/

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.oauth2.server.authorization.config.annotation.web.configuration.OAuth2AuthorizationServerConfiguration;

@EnableWebSecurity

public class AuthorizationServerConfigurer {

public void configure(HttpSecurity http) throws Exception {

OAuth2AuthorizationServerConfiguration.applyDefaultSecurity(http);

http.authorizeRequests().anyRequest().authenticated();

}

}

/\*

AuthorizationServerConfigurer sets up default OAuth2 authorization server endpoints and security.

\*/

/\*

8. AuthorizationGrantType

\*/

/\*

AuthorizationGrantType is used while defining the RegisteredClient. Examples:

- AUTHORIZATION\_CODE: Standard user login flow.

- REFRESH\_TOKEN: Getting a new token without logging in again.

\*/

/\*

9. OAuth2AccessToken

\*/

import org.springframework.security.oauth2.core.OAuth2AccessToken;

OAuth2AccessToken accessToken = new OAuth2AccessToken(OAuth2AccessToken.TokenType.BEARER, "access-token-value", Instant.now(), Instant.now().plusSeconds(3600));

/\*

OAuth2AccessToken represents the token used to access secure resources.

\*/

/\*

10. OAuth2RefreshToken

\*/

import org.springframework.security.oauth2.core.OAuth2RefreshToken;

OAuth2RefreshToken refreshToken = new OAuth2RefreshToken("refresh-token-value", Instant.now(), Instant.now().plusSeconds(7200));

/\*

OAuth2RefreshToken allows generating a new access token without re-authentication.

\*/