Microservice Security

I want to understand complete flow of OAuth 2.0 like from token fetching at client side and passing it to service then how this token gets validated from service side give me code example of client and service provider spring boot project

**ChatGPT said:**

ChatGPT

The OAuth 2.0 flow is a widely-used standard for token-based authentication. It involves obtaining an access token from an Authorization Server, which is then passed with API requests to the resource server (service provider) for access. Let’s walk through the **OAuth 2.0 Client Credentials Flow**, one of the commonly used flows for server-to-server communication (machine-to-machine).

**Complete OAuth 2.0 Flow:**

1. **Client Requests Token**:
   * The client makes a POST request to the Authorization Server’s /token endpoint.
   * The Authorization Server authenticates the client using credentials (client ID and secret).
   * The Authorization Server returns an access token to the client.
2. **Client Uses Access Token**:
   * The client includes the access token in the Authorization header when making requests to the Resource Server (API).
3. **Resource Server Validates the Token**:
   * The Resource Server (API) validates the access token with the Authorization Server (or introspects it locally).
   * If the token is valid, access is granted to the requested resource.

Now, I'll walk you through how to implement this with **Spring Boot**.

**1. Authorization Server (Spring Authorization Server)**

Here’s a simple Spring Boot project that acts as the **Authorization Server**:

**pom.xml dependencies:**

Add the following dependencies for Spring Security and OAuth2 authorization server:

xml

Copy code

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-oauth2-authorization-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

**OAuth2 Authorization Server Configuration:**

Create an authorization server configuration to expose the /token endpoint.

java

Copy code

@Configuration

@EnableAuthorizationServer

public class AuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {

@Override

public void configure(ClientDetailsServiceConfigurer clients) throws Exception {

clients.inMemory()

.withClient("client-id")

.secret(passwordEncoder().encode("client-secret"))

.authorizedGrantTypes("client\_credentials")

.scopes("read", "write")

.accessTokenValiditySeconds(3600); // 1 hour validity

}

@Override

public void configure(AuthorizationServerEndpointsConfigurer endpoints) {

endpoints.tokenStore(tokenStore());

}

@Bean

public TokenStore tokenStore() {

return new InMemoryTokenStore(); // For simplicity, using in-memory token store

}

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

}

This configuration sets up an in-memory client with a **client ID** of client-id and a **secret** of client-secret. The Authorization Server will issue access tokens when this client requests one via the client\_credentials grant type.

**Security Configuration:**

java

Copy code

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.csrf().disable()

.authorizeRequests()

.anyRequest().authenticated()

.and()

.httpBasic(); // Basic auth for client\_id and client\_secret

}

}

**2. Client Application (OAuth2 Client)**

The client application requests a token from the authorization server and uses that token to call the resource server.

**pom.xml dependencies:**

Add the following dependencies for Spring Web and OAuth2 Client:

xml

Copy code

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-oauth2-client</artifactId>

</dependency>

**RestTemplate with OAuth2 Token Fetching:**

java

Copy code

@Component

public class OAuth2Client {

private final RestTemplate restTemplate;

public OAuth2Client(RestTemplateBuilder restTemplateBuilder) {

this.restTemplate = restTemplateBuilder.build();

}

public String getAccessToken() {

String tokenUrl = "http://localhost:8080/oauth/token";

HttpHeaders headers = new HttpHeaders();

headers.setBasicAuth("client-id", "client-secret"); // Client credentials

MultiValueMap<String, String> body = new LinkedMultiValueMap<>();

body.add("grant\_type", "client\_credentials");

HttpEntity<MultiValueMap<String, String>> request = new HttpEntity<>(body, headers);

ResponseEntity<Map> response = restTemplate.exchange(tokenUrl, HttpMethod.POST, request, Map.class);

return response.getBody().get("access\_token").toString();

}

public String callApiWithToken(String token) {

String apiUrl = "http://localhost:8081/api/resource";

HttpHeaders headers = new HttpHeaders();

headers.setBearerAuth(token);

HttpEntity<String> entity = new HttpEntity<>(null, headers);

ResponseEntity<String> response = restTemplate.exchange(apiUrl, HttpMethod.GET, entity, String.class);

return response.getBody();

}

}

**Client Application Flow:**

1. The getAccessToken() method requests an access token from the Authorization Server.
2. The callApiWithToken() method makes an API call with the retrieved token.

**3. Resource Server (API)**

The resource server will validate the token with the Authorization Server before granting access to the requested resource.

**pom.xml dependencies:**

Add the following dependencies for Spring Security OAuth2 Resource Server:

xml

Copy code

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-oauth2-resource-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

**Resource Server Configuration:**

java

Copy code

@Configuration

@EnableWebSecurity

public class ResourceServerConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.authorizeRequests()

.antMatchers("/api/\*\*").authenticated()

.and()

.oauth2ResourceServer()

.jwt(); // Enable JWT token validation

}

}

**Exposing a Secured Resource:**

java

Copy code

@RestController

@RequestMapping("/api")

public class ApiController {

@GetMapping("/resource")

public String getProtectedResource() {

return "This is a protected resource";

}

}

**4. Run the Flow:**

1. **Start Authorization Server**: Run the Spring Boot app that contains the Authorization Server.
2. **Start Resource Server**: Run the Spring Boot app that contains the Resource Server.
3. **Run Client Application**:
   * Call the OAuth2Client.getAccessToken() method to retrieve the access token from the Authorization Server.
   * Use the OAuth2Client.callApiWithToken(token) method to make a request to the Resource Server, passing the token in the Authorization header.

**Example Flow:**

1. Client sends a POST request to /token endpoint to get an access token.
2. Client uses the access token in the Authorization header to access a protected resource.
3. Resource server validates the access token and serves the request if valid.

**Summary:**

* **Authorization Server**: Issues tokens (in this case, using the client credentials grant).
* **Resource Server**: Protects APIs and validates tokens.
* **Client**: Fetches tokens from the Authorization Server and uses them to access the Resource Server.

This is a basic OAuth 2.0 client credentials flow. You can extend it by adding more features like token introspection, refresh tokens, or different grant types.

4o

**You said:**

why do OAuth 2.0 have 'Authorization Server' concept why dont we say Authentication Server

**ChatGPT said:**

ChatGPT

In OAuth 2.0, the term **"Authorization Server"** is used deliberately to distinguish between **authorization** and **authentication**, as these are two distinct concepts. Here's a breakdown of the difference and why the terminology exists:

**1. Authorization vs. Authentication:**

* **Authentication**: This process is about verifying the identity of the user or client. It answers the question, "Who are you?" Authentication involves ensuring that the user is who they claim to be, typically through mechanisms like username/password, biometrics, tokens, etc.
* **Authorization**: This process is about granting permissions. It answers the question, "What can you do?" Authorization determines what resources or actions the user is allowed to access or perform after they've been authenticated.

**2. OAuth 2.0 is Primarily About Authorization:**

OAuth 2.0 is designed to **grant access (authorization)** to protected resources without sharing user credentials (like passwords). It’s mainly focused on giving clients (such as web apps, mobile apps, or APIs) delegated permission to act on behalf of a user.

* **Authorization Server** is responsible for **issuing access tokens** that allow the client to access resources on behalf of the user, but it doesn't authenticate users directly (at least, not as its primary purpose).

For example, when you log in via Google OAuth (using your Google account to access a third-party app), Google’s OAuth service is acting as an **authorization server** that issues a token to the third-party app, allowing it to perform certain actions on your behalf (like accessing your email contacts), but it doesn't primarily serve to authenticate the app itself.

**3. Why Not Call It "Authentication Server"?**

OAuth 2.0 does not handle authentication directly, which is why the term "Authentication Server" isn't used. The role of verifying identity (authentication) is often done by a separate mechanism (e.g., username/password, Single Sign-On, etc.).

* **OpenID Connect (OIDC)**, which is built on top of OAuth 2.0, **adds authentication** on top of OAuth 2.0. In OpenID Connect, the **identity of the user** is verified, and the client receives an ID token along with the access token, serving as proof of authentication. This is why OpenID Connect is often used when you need authentication (login) in addition to authorization.

So, while OAuth 2.0 can be used as part of a broader system that includes authentication, its core focus is on **authorization**, which is why we refer to it as an **Authorization Server**.

**4. Separation of Concerns:**

* **Authorization Server** issues tokens and grants permissions (authorizations) based on client identity or user consent.
* **Authentication Mechanism** (could be integrated or separate) handles the process of verifying the user’s identity, and this may or may not be part of the OAuth 2.0 flow (depending on the implementation and whether OpenID Connect is layered on top).

**Example of the Difference:**

* **OAuth 2.0 without Authentication**: If you have an API service that trusts certain clients (e.g., machine-to-machine communication), it might issue tokens without authenticating any end user. Here, the focus is only on authorizing what the client can do, not who the user is.
* **OAuth 2.0 with Authentication (via OpenID Connect)**: When a user logs into a third-party app using their Google credentials, Google’s OpenID Connect flow authenticates the user and then issues an ID token (authentication) alongside the OAuth access token (authorization). The ID token proves who the user is, while the access token grants permission to access their resources.

**Conclusion:**

OAuth 2.0 is focused on **authorization**, i.e., granting permission to clients to access resources on behalf of users, which is why the term **"Authorization Server"** is used. The responsibility of **authentication** is usually handled outside of OAuth 2.0 or layered on top of it (like in OpenID Connect). Therefore, calling it an "Authentication Server" would not fully capture its intended purpose in the OAuth framework.

4o

**You said:**

ok then in spring boot @PreAuthorize annotation api level what will i say this Authentication or authorization it is authorization means we are doing two time authorization one with OAuth 2.0 and other using spring annotations

**ChatGPT said:**

ChatGPT

In Spring Boot, the @PreAuthorize annotation is primarily concerned with **authorization** rather than authentication. Here's a breakdown of how it fits into the broader context of your application, especially when using OAuth 2.0 alongside Spring Security:

**Understanding @PreAuthorize**

1. **Purpose**:
   * The @PreAuthorize annotation is used to secure methods by specifying authorization rules. It checks whether the authenticated user has the necessary permissions to execute the method.
2. **Authorization vs. Authentication**:
   * **Authorization**: @PreAuthorize checks if the currently authenticated user (who has been authenticated by some means, such as OAuth 2.0) has the correct roles or permissions to access a particular resource or execute a method.
   * **Authentication**: This is the process of verifying the user's identity (e.g., through OAuth 2.0), which typically occurs before the @PreAuthorize check. For example, a user logs in through an OAuth 2.0 flow, and after successfully obtaining an access token, they become an authenticated user.

**Two Layers of Authorization**

When using OAuth 2.0 with Spring Security, there are effectively two layers of authorization:

1. **OAuth 2.0 Authorization**:
   * The user is authenticated via the OAuth 2.0 flow, receiving an access token that grants them permissions to access certain resources. At this point, the authorization server (e.g., Google, Auth0) determines what resources the user can access based on their roles and permissions.
2. **Spring Security Authorization**:
   * After authentication, when a request is made to your Spring Boot application, the @PreAuthorize annotation evaluates the user's roles or authorities against the method's requirements. This is where you can specify fine-grained access control based on your application's business logic.

**Example**

Here’s how you might see it in action:

java

Copy code

@RestController

@RequestMapping("/api")

public class MyController {

@PreAuthorize("hasRole('ROLE\_USER')")

@GetMapping("/user")

public ResponseEntity<String> getUserData() {

return ResponseEntity.ok("User data");

}

@PreAuthorize("hasRole('ROLE\_ADMIN')")

@GetMapping("/admin")

public ResponseEntity<String> getAdminData() {

return ResponseEntity.ok("Admin data");

}

}

**Conclusion**

* The @PreAuthorize annotation is purely about **authorization**; it determines whether an authenticated user has the right to access a particular resource or perform a specific action.
* While OAuth 2.0 manages user **authentication** and grants access tokens, Spring Security’s @PreAuthorize manages **authorization** within your application based on the authenticated user's roles or permissions.

So, in the context you provided, yes, you are effectively performing **two layers of authorization**:

* The first is at the OAuth 2.0 level (to obtain the access token).
* The second is at the Spring Security level (to enforce method-level security with @PreAuthorize).

Both layers work together to ensure that users not only authenticate successfully but also have the appropriate permissions to access specific resources in your application.

4o mini

**You said:**

how this role authorization value is passed to service from resttemplate client so spring could understand

**ChatGPT said:**

ChatGPT

When using OAuth 2.0 for authorization in a Spring Boot application, roles or authorities assigned to a user are typically conveyed through a **JWT (JSON Web Token)** or a similar token format. This token is generated by the **Authorization Server** after the user successfully authenticates and is then passed along with each request made by the client (often using RestTemplate or similar HTTP clients).

Here’s how the flow works step-by-step, including how to pass role authorization values and how Spring Security understands them.

**1. User Authentication and Token Generation**

1. **User Authenticates**: The user logs in through an OAuth 2.0 provider (e.g., Google, Auth0).
2. **Token Issued**: Upon successful authentication, the Authorization Server issues an access token (often a JWT) that encodes user roles and permissions. This token is then sent back to the client application.

**2. Client Sending Requests with the Access Token**

When the client application (let’s say a Spring Boot application) makes a request to a protected resource, it includes the access token in the **Authorization** header:

java

Copy code

import org.springframework.http.HttpEntity;

import org.springframework.http.HttpHeaders;

import org.springframework.http.ResponseEntity;

import org.springframework.web.client.RestTemplate;

public class MyClient {

private RestTemplate restTemplate;

private String accessToken; // Assume this token is obtained after login

public MyClient(RestTemplate restTemplate, String accessToken) {

this.restTemplate = restTemplate;

this.accessToken = accessToken;

}

public ResponseEntity<String> getProtectedResource() {

// Set the Authorization header

HttpHeaders headers = new HttpHeaders();

headers.set("Authorization", "Bearer " + accessToken);

HttpEntity<String> entity = new HttpEntity<>(headers);

return restTemplate.exchange("http://your-service/api/protected", HttpMethod.GET, entity, String.class);

}

}

**3. Server-Side Token Validation and Role Extraction**

1. **Filter**: When the request reaches the server-side Spring application, a filter (typically a OncePerRequestFilter) checks for the presence of the Authorization header.
2. **Token Validation**: The filter extracts the token and validates it. If the token is a JWT, it can be parsed to extract claims, including user roles or authorities.
3. **Setting Security Context**: If the token is valid, the user’s authorities are set in the Spring Security SecurityContextHolder. This way, Spring Security knows which roles are associated with the authenticated user.

Here’s how you might see this implemented in a Spring Boot application:

java

Copy code

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.security.core.Authentication;

import org.springframework.security.oauth2.server.resource.authentication.JwtAuthenticationToken;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class MyProtectedController {

@GetMapping("/api/protected")

public String getProtectedData() {

// Retrieve the authentication object

Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

// Check if it's a JWT token

if (authentication instanceof JwtAuthenticationToken) {

JwtAuthenticationToken jwtAuth = (JwtAuthenticationToken) authentication;

// Extract roles/authorities from the JWT

return "Authenticated as: " + jwtAuth.getPrincipal() + " with roles: " + jwtAuth.getAuthorities();

}

return "Unauthorized";

}

}

**4. Role Authorization Logic**

With the roles now available in the SecurityContext, you can use @PreAuthorize, method security annotations, or configure HTTP security rules to enforce access control based on roles:

java

Copy code

import org.springframework.security.access.prepost.PreAuthorize;

@RestController

@RequestMapping("/api")

public class MyController {

@PreAuthorize("hasRole('ROLE\_USER')")

@GetMapping("/user")

public String getUserData() {

return "User data accessed.";

}

@PreAuthorize("hasRole('ROLE\_ADMIN')")

@GetMapping("/admin")

public String getAdminData() {

return "Admin data accessed.";

}

}

**Summary**

* **Token Generation**: The access token, often a JWT, carries information about the user's roles and permissions.
* **Request with Token**: The client includes the token in the Authorization header when making requests.
* **Server-Side Validation**: The Spring Boot application validates the token, extracts roles, and sets them in the security context.
* **Authorization Logic**: Role checks can then be performed using Spring Security annotations or configuration.

This setup allows for a secure and scalable way to handle role-based access control in your Spring Boot application, leveraging OAuth 2.0 for authentication and authorization.

4o mini

**You said:**

it means roles is defined at Authorization Server size and included in json token or elsewhere

**ChatGPT said:**

ChatGPT

Yes, that's correct! In the OAuth 2.0 flow, roles and permissions are typically defined and managed at the **Authorization Server** level. Here's how this works in detail:

**1. Roles and Permissions Defined at Authorization Server**

* **User Management**: The Authorization Server (such as Keycloak, Auth0, or your own implementation) is responsible for managing user accounts, roles, and permissions. This includes defining what roles exist (e.g., ROLE\_USER, ROLE\_ADMIN) and which users are assigned to those roles.
* **Token Structure**: When a user successfully authenticates, the Authorization Server issues an **access token**. If the access token is a JWT (JSON Web Token), it can contain claims that represent the user's roles and permissions.

**2. JWT Structure and Claims**

A typical JWT consists of three parts: the header, the payload, and the signature. The **payload** part is where the claims are stored, and it may look something like this:

json

Copy code

{

"sub": "1234567890", // Subject (user identifier)

"name": "John Doe", // User name

"iat": 1516239022, // Issued at

"exp": 1516242622, // Expiration time

"roles": ["ROLE\_USER", "ROLE\_ADMIN"] // User roles

}

**3. Token Issuance Process**

* **Authentication Flow**: When a user logs in, the Authorization Server verifies the user's credentials.
* **Token Generation**: Upon successful authentication, it generates a token that includes relevant claims, such as the user's roles.
* **Token Delivery**: The token is then sent back to the client application, which will use it for subsequent requests.

**4. Using Roles in the Client Application**

* **Sending the Token**: The client application (e.g., a Spring Boot app) sends the access token as part of the Authorization header in requests to the Resource Server (the backend service).
* **Validation and Role Extraction**: The Resource Server validates the token, extracts the roles from the claims, and sets them in the Spring Security context.

**Example of Role Extraction**

Here's how you might extract roles from a JWT in a Spring Boot application:

java

Copy code

import org.springframework.security.core.Authentication;

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.security.oauth2.server.resource.authentication.JwtAuthenticationToken;

@RestController

public class MyController {

@GetMapping("/api/protected")

public String getProtectedData() {

Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

if (authentication instanceof JwtAuthenticationToken) {

JwtAuthenticationToken jwtAuth = (JwtAuthenticationToken) authentication;

// Extract roles from the JWT token

List<String> roles = jwtAuth.getAuthorities().stream()

.map(GrantedAuthority::getAuthority)

.collect(Collectors.toList());

return "User roles: " + roles;

}

return "Unauthorized";

}

}

**Summary**

* **Authorization Server Role Management**: Roles are defined and managed at the Authorization Server.
* **JWT Claims**: These roles are typically included in the JWT payload as claims.
* **Client Usage**: The client application uses the JWT for authentication, and the Resource Server validates the token and extracts roles for authorization checks.

By managing roles at the Authorization Server, you centralize user management and simplify the security architecture of your application. This approach allows for flexible and scalable security configurations across different services.