<https://datatracker.ietf.org/doc/html/rfc6749> -🡪 uemy course content

In my previous Project Tesco, we are using client credentails grant type for micro service, service to service communications.

**In microservices client credentials grant types, how it works?**

**ChatGPT said:**

Great question 👍 Let’s break it down clearly.

**Client Credentials Grant Flow in Microservices**

The **client credentials grant type** is part of OAuth2.0, used when one service (microservice A) needs to talk to another service (microservice B) securely **without a user being involved**.  
Example: An **Order Service** fetching product details from **Product Service**.

**How it works step by step**

1. **Service Registers as a Client**
   * Each microservice that wants to call another service must be registered with the **Authorization Server** (e.g., Keycloak, Okta, Azure AD).
   * During registration, it gets:
     + client\_id
     + client\_secret
2. **Requesting a Token**
   * When **Order Service** needs to call **Product Service**, it first requests an **access token** from the **Authorization Server**.
   * It sends:
   * POST /oauth2/token
   * Content-Type: application/x-www-form-urlencoded
   * grant\_type=client\_credentials
   * client\_id=order-service
   * client\_secret=order-secret
   * No user credentials here, only service credentials.
3. **Authorization Server Responds**
   * The Auth server validates the credentials.
   * If valid, it issues a **JWT access token** (or opaque token):
   * {
   * "access\_token": "eyJhbGciOiJIUzI1NiIsInR5...",
   * "token\_type": "Bearer",
   * "expires\_in": 3600,
   * "scope": "read:products"
   * }
4. **Calling the Target Microservice**
   * Order Service includes this token in the HTTP header:
   * GET /products/123
   * Host: product-service
   * Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5...
5. **Target Service Validates Token**
   * **Product Service** validates the JWT:
     + Checks **signature** (with public key of Auth Server).
     + Checks **expiration time**.
     + Checks **audience / scopes**.
   * If valid → grants access.
   * If invalid → responds with 401 Unauthorized.

**Token Validation**

There are 2 main strategies for validation:

* **Local validation**:
  + If the token is JWT, Product Service can validate it using Auth Server’s **public key** without calling the Auth server again.
  + Fast and decentralized.
* **Introspection endpoint**:
  + If the token is opaque, Product Service calls the **Auth Server introspection endpoint** to check validity.

**Advantages of Client Credentials Grant**

✅ No user interaction needed  
✅ Good for **machine-to-machine** communication  
✅ Secure and standardized with OAuth2.0  
✅ Scopes allow fine-grained permissions (e.g., "order-service can only read products, not delete")

**During each service registered with the oauth server we will get??**

**ChatGPT said:**

Ahh, I see where you’re going 👍 You’re asking:

👉 **When a microservice registers with the OAuth (Authorization) Server, what exactly do we get?**

Let’s make it crystal clear.

**🔐 When a service registers with the OAuth server, it gets:**

1. **Client ID**
   * A unique identifier for the service (like a username).
   * Example:
   * client\_id = order-service
2. **Client Secret**
   * A password for the service (only known to this service + Auth Server).
   * Example:
   * client\_secret = x9f8h23kjsdf9823kjhf9
3. **Scopes / Permissions**
   * Defines what this service is allowed to do.
   * Example:
     + read:products → can read product details
     + write:orders → can create orders
4. **Grant Type(s) Allowed**
   * Specifies how the service can ask for tokens.
   * For machine-to-machine → client\_credentials grant type.

**⚡ What about public keys?**

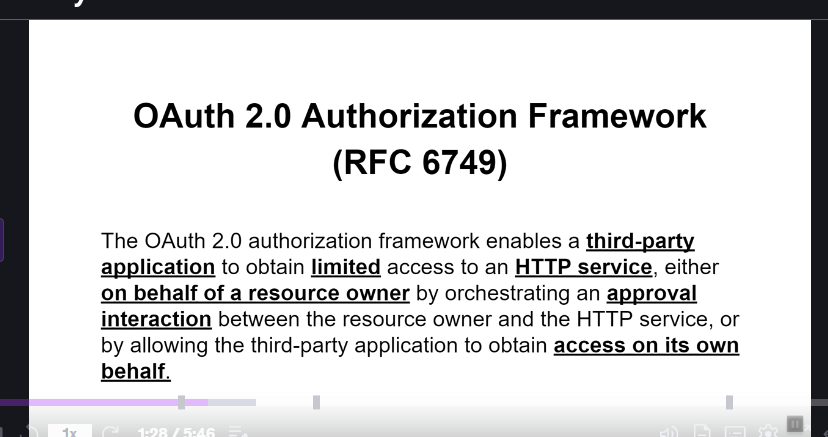
* **Important:**  
  When you register a service with the OAuth server, you **do not directly get a public key**.
* Public/Private keys belong to the **Authorization Server itself** (used to sign JWTs).
* Services (like Product Service) only **fetch the public key** later from the JWKS endpoint when they need to validate tokens.

✅ **So registration gives you:**

* client\_id
* client\_secret
* (optionally) assigned scopes/roles
* allowed grant types

❌ It does **not** give you the JWT public key. That comes from the Auth server’s published JWKS.

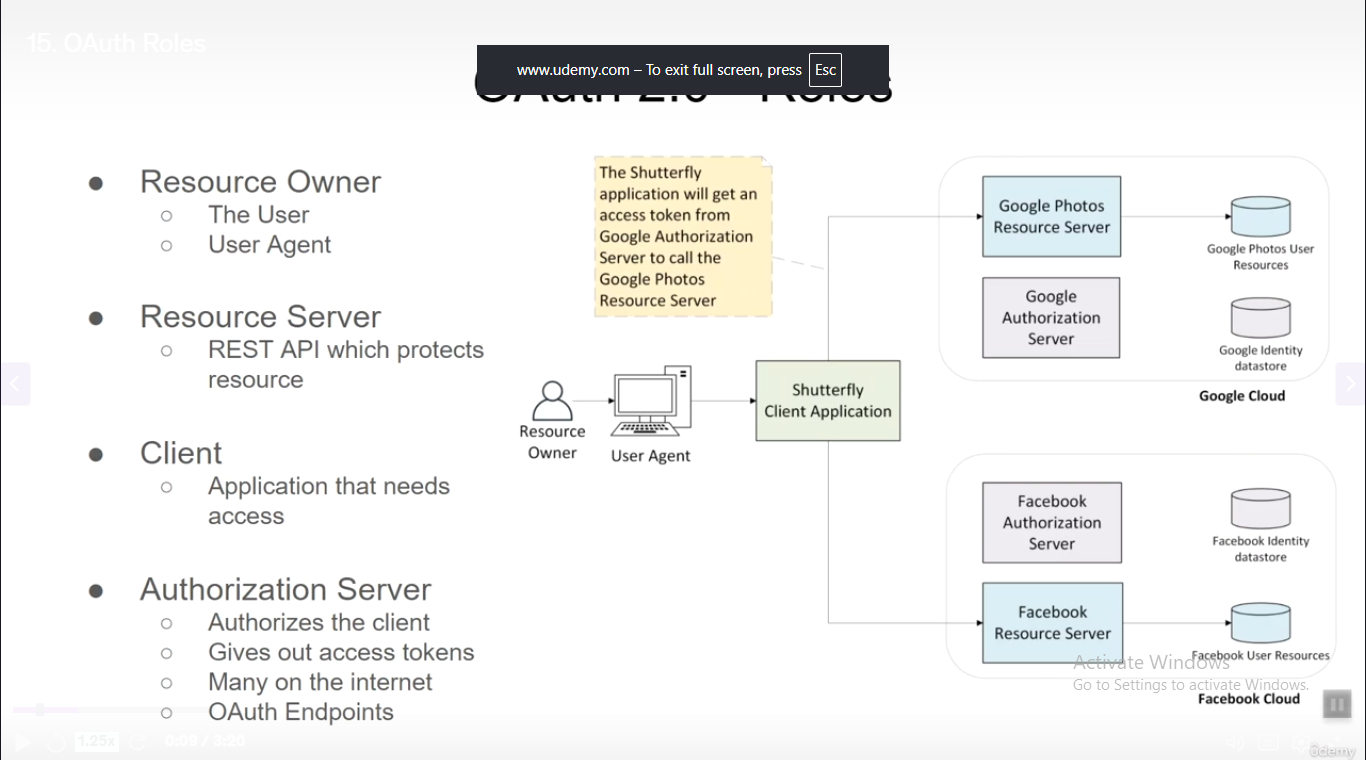
What is Oauth2



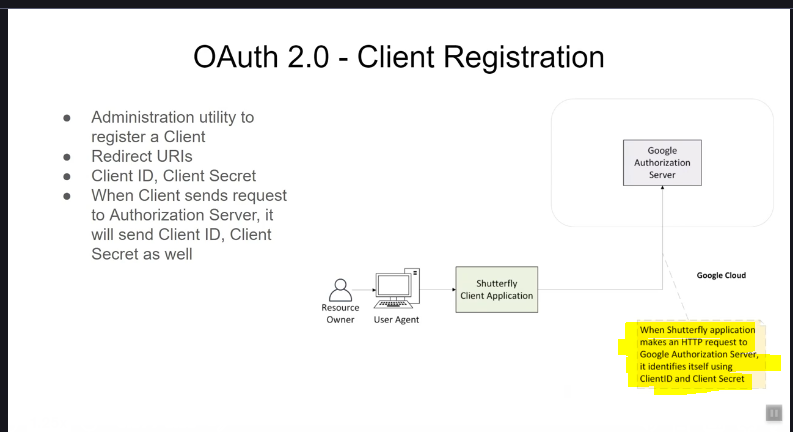
I can login to udemy by using gmail account

Here client is Udemy, Oauth2 is server Google, resource owner is me and resource service is also Google.

The **Authorization**in OAuth (stands for Open Authorization) refers to the "**permission**" given by the user to the Third Party application to act on his or her behalf. In the **Shutterfly**demonstration, I clicked on the **Allow** button to give permission to the Shutterfly application to use my Google Photos.



Oauth2 Client Registration.



**Suppose I developed a application which needs to be login by gmail. For this client registration steps with google.**

suppose my client is XYZ-Ramu , now explain same with this ?

**🧩 Scenario:**

You have a website called:

https://xyz-ramu.com

You want users to **log in with Google**.

You register your app with Google and get:

* client\_id: abc123-xyz.apps.googleusercontent.com
* client\_secret: super-secret-value
* redirect\_uri: https://xyz-ramu.com/oauth2/callback

**✅ Step-by-Step OAuth2 Flow with Google**

**🟢 STEP 1: You Register Your App (One-time Setup)**

At https://console.cloud.google.com:

* Create a new project called XYZ-Ramu
* Enable **Google OAuth API**
* Register OAuth client (Web Application)
* Set redirect\_uri to:

https://xyz-ramu.com/oauth2/callback

✅ You get:

client\_id: abc123-xyz.apps.googleusercontent.com

client\_secret: super-secret-value

**🟡 STEP 2: User Clicks “Login with Google” on Your Website**

You redirect the browser to Google **Authorization Endpoint**:

http

CopyEdit

GET https://accounts.google.com/o/oauth2/v2/auth?

client\_id=abc123-xyz.apps.googleusercontent.com

&redirect\_uri=https://xyz-ramu.com/oauth2/callback

&response\_type=code

&scope=openid email profile

✅ The **browser is redirected to Google**.  
Google asks the user to **login** and **approve your app**.

**🟠 STEP 3: Google Redirects Back to Your Website**

After successful login, Google **redirects the browser back to your app**:

http

CopyEdit

https://xyz-ramu.com/oauth2/callback?code=XYZ\_CODE\_123

✅ The code is **temporary** and **valid for only a few minutes**.  
You **cannot reuse it**.

**🔵 STEP 4: Your Server (Backend) Sends Code to Google**

Now, your **backend (not frontend)** sends a secure POST request to Google:

http

CopyEdit

POST https://oauth2.googleapis.com/token

Content-Type: application/x-www-form-urlencoded

client\_id=abc123-xyz.apps.googleusercontent.com

&client\_secret=super-secret-value

&code=XYZ\_CODE\_123

&redirect\_uri=https://xyz-ramu.com/oauth2/callback

&grant\_type=authorization\_code

✅ This exchange happens **securely on the server**  
❗️ **Do NOT** do this in frontend — the client\_secret must stay private!

**🟣 STEP 5: Google Responds with Tokens**

Google replies with:

json

CopyEdit

{

"access\_token": "ya29.a0Af...",

"id\_token": "eyJhbGciOiJIUzI1...",

"refresh\_token": "1//0gk0...",

"expires\_in": 3599,

"token\_type": "Bearer"

}

✅ You use:

* access\_token to call Google APIs (like Gmail, Drive)
* id\_token to authenticate the user in your system (decode this JWT)
* refresh\_token to get a new access\_token later

**🔴 STEP 6: User is Now Logged In**

Now you can:

* Create a user session
* Show the user’s name, email, profile picture
* Save user info in your DB if needed

**📌 Final Summary (Clean Table)**

| **Step** | **Who?** | **What Happens** |
| --- | --- | --- |
| 1️⃣ Register App | You | On Google Cloud, get client\_id + client\_secret |
| 2️⃣ Redirect User to Google | Frontend | Browser goes to Google login screen |
| 3️⃣ Google Redirects Back | Google → Browser | Sends code to https://xyz-ramu.com/oauth2/callback?code=... |
| 4️⃣ Exchange Code | Your Backend | POSTs code to Google with secret |
| 5️⃣ Get Tokens | Google | Returns access\_token, id\_token, etc. |
| 6️⃣ User Logged In | Your App | Authenticated and authorized |

………………………………………………………………………………………………………………………………………………

and also explain which is client, resource server and oauth server from this example and redirect\_uri=https://xyz-ramu.com/oauth2/callback why do we use this url?

## Real-World Scenario: XYZ-Ramu with Google Login

### 🧩 Let's define what each part means in your actual case.

| **Role** | **Real-World Example** | **What It Does** |
| --- | --- | --- |
| ✅ **Client** | Your app: XYZ-Ramu | The app that wants to access the user’s data. |
| 👤 **Resource Owner** | The user (e.g. John) | The person who is logging in, whose data you want to access (email, name, etc.). |
| 🔐 **Authorization Server** | accounts.google.com (Google Login Page) | Google’s server that handles user login and issues tokens. |
| 📦 **Resource Server** | Google APIs (e.g., https://www.googleapis.com/userinfo) | The server that holds the user's data like profile info, email, etc. |

## 🎯 What XYZ-Ramu wants to do:

Your app wants to **let users log in using Google**, and then get their **profile info** (name, email, picture).

## 🔁 Full Flow with All Roles Explained

### 🔵 Step 1: User clicks ****“Login with Google”****

🔹 (From your website: https://xyz-ramu.com)  
🔹 Your app redirects the browser to:

http

CopyEdit

GET https://accounts.google.com/o/oauth2/v2/auth?

client\_id=abc123...

&redirect\_uri=https://xyz-ramu.com/oauth2/callback

&response\_type=code

&scope=openid email profile

| **Who is involved?** | **What’s happening?** |
| --- | --- |
| **Client = XYZ-Ramu** | Makes the request |
| **Authorization Server = Google** | Receives request and shows login screen |
| **User = Resource Owner** | Logs in and approves access |

### 🔵 Step 2: Google redirects back to your app

After user logs in, Google sends:

bash

CopyEdit

https://xyz-ramu.com/oauth2/callback?code=xyz-code-abc

This code is a **temporary ticket** your app can exchange for tokens.

### 🔵 Step 3: Your App (Server) sends code to Google

Your backend (Spring Boot) sends a **secure POST** to:

bash

CopyEdit

POST https://oauth2.googleapis.com/token

With this:

x

CopyEdit

code=xyz-code-abc

client\_id=abc123...

client\_secret=shhh-secret

redirect\_uri=https://xyz-ramu.com/oauth2/callback

grant\_type=authorization\_code

| **Who is involved?** | **What’s happening?** |
| --- | --- |
| **Client = XYZ-Ramu backend** | Sends request to exchange code |
| **Authorization Server = Google** | Validates and responds with tokens |

### 🔵 Step 4: Google responds with tokens

Example response:

json

CopyEdit

{

"access\_token": "ya29.abc...",

"id\_token": "eyJhbGci...",

"expires\_in": 3600,

"token\_type": "Bearer"

}

✅ You now have an **access token** to call Google's APIs.  
✅ You can also decode the **ID token (JWT)** to get user info.

### 🔵 Step 5: Your App calls Google API to get user info

http

CopyEdit

GET https://www.googleapis.com/oauth2/v3/userinfo

Authorization: Bearer ya29.abc...

Google returns:

json

CopyEdit

{

"email": "john@example.com",

"name": "John",

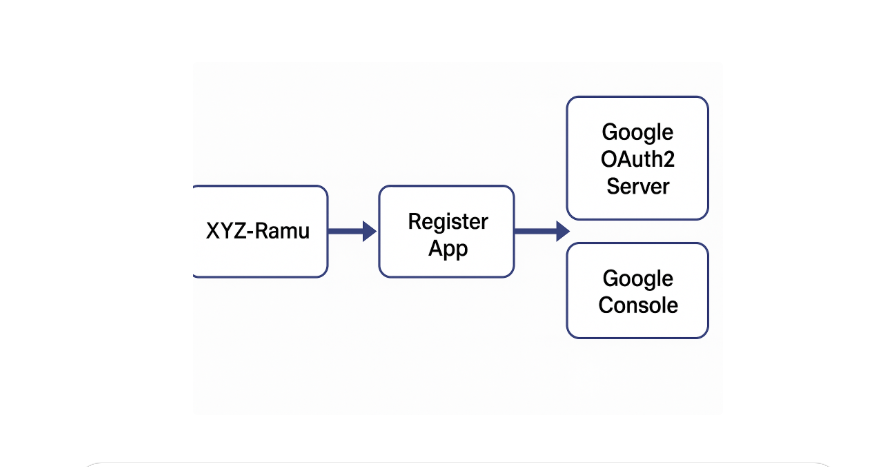
"picture": "https://..."

}

| **Who is involved?** | **What’s happening?** |
| --- | --- |
| **Client = XYZ-Ramu** | Makes request to get user profile |
| **Resource Server = Google API** | Returns user info |
| **Authorization Server** | Not involved in this step anymore |

## ✅ FINAL SUMMARY TABLE

| **Role** | **Entity** | **Example** |
| --- | --- | --- |
| **Client** | Your App | https://xyz-ramu.com |
| **Resource Owner** | User | A person logging in |
| **Authorization Server** | Google OAuth | https://accounts.google.com |
| **Resource Server** | Google APIs | https://www.googleapis.com/... |
| **redirect\_uri** | Your App's Callback | https://xyz-ramu.com/oauth2/callback (where Google sends the code) |



## Common response\_type Values

| **response\_type** | **Use Case** | **Description** |
| --- | --- | --- |
| code | **Authorization Code Flow** | Secure, server-side flow. App receives an **authorization code**, then exchanges it for tokens. Requires client\_secret. ✅ Recommended for web apps and Spring Boot |
| token | **Implicit Flow (Legacy)** | Directly returns an **access token** in the redirect URL. No backend required. ⚠️ Not secure, **not recommended anymore** |
| id\_token | **OIDC Implicit Flow** | Returns just an **ID token** (user identity). Used for authentication in front-end apps |
| id\_token token | **OIDC Implicit Flow** | Returns both id\_token and access\_token directly in URL. Frontend apps (not secure for production) |
| code id\_token | **Hybrid Flow** | Combines code and id\_token (gets ID token immediately + can later exchange code for access token) |
| code token | **Hybrid Flow** | Gets both code and access\_token |
| code id\_token token | **Hybrid Flow (All-in-One)** | Gets everything — code, id token, and access token |

### ✅ Recommended (for most secure apps like Spring Boot):

http

CopyEdit

response\_type=code

Use this with:

* client\_id
* redirect\_uri
* scope=openid email profile
* state (optional but recommended)
* Exchange code later for tokens using client\_secret

Would you like a diagram comparing the flows?

##### You said:

different scopes?

###### ChatGPT said:

Excellent! Let's explore the **different OAuth2 and OpenID Connect scopes** — what they mean and when to use them.

## ✅ What is a scope?

* The scope parameter tells the **Authorization Server** (e.g., Google) what **permissions** your app is asking for.
* Scopes define:
  + What data your app can access
  + What actions your app can perform on behalf of the user

## 🔹 Common Scopes (OAuth2 + OpenID Connect)

### 🔐 OpenID Connect Scopes (for Authentication)

| **Scope** | **Purpose** |
| --- | --- |
| openid | Enables OpenID Connect (required to get id\_token) |
| email | Access user's email address |
| profile | Access user's basic profile info (name, picture, etc.) |
| address | Access user's physical address |
| phone | Access user's phone number |

✅ Minimum for login with Google:

http

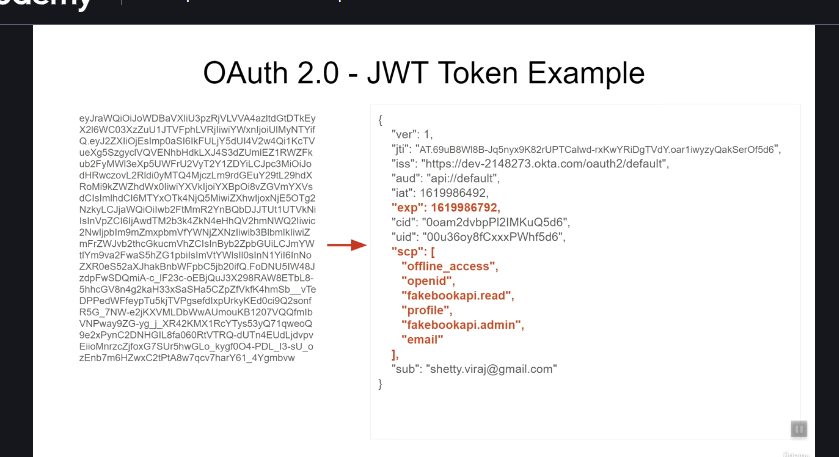
CopyEdit

scope=openid email profile

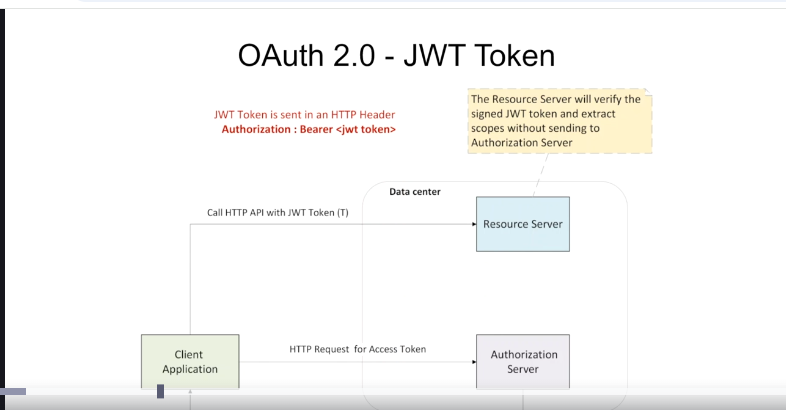
### 🛠️ Google API Scopes (for Accessing User Data)

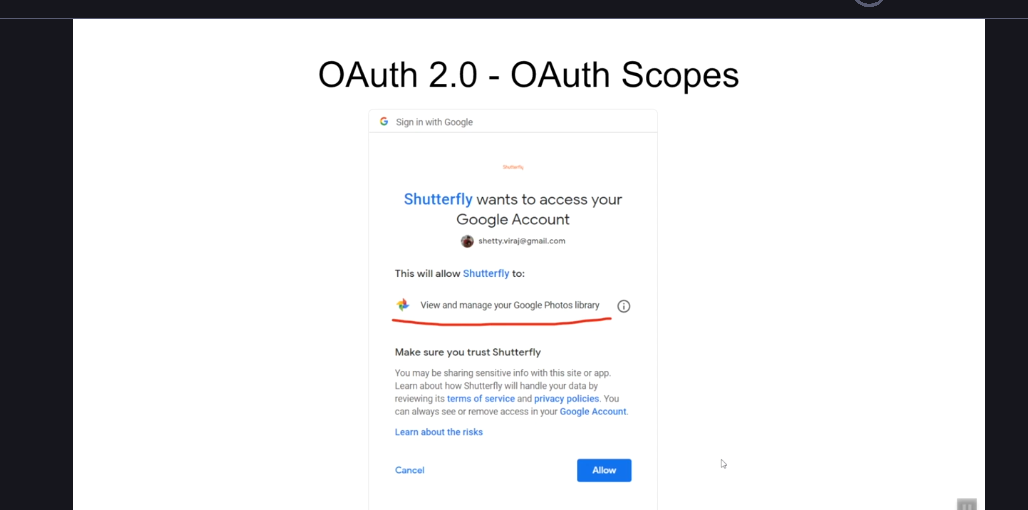
| **Scope** | **Description** |
| --- | --- |
| https://www.googleapis.com/auth/userinfo.email | Access user’s email |
| https://www.googleapis.com/auth/userinfo.profile | Access user’s full Google profile |
| https://www.googleapis.com/auth/calendar.readonly | Read user’s Google Calendar |
| https://www.googleapis.com/auth/drive.file | Access user's Google Drive files created by your app |
| https://www.googleapis.com/auth/gmail.readonly | Read user's Gmail |
| https://www.googleapis.com/auth/youtube.readonly | Read user's YouTube data |

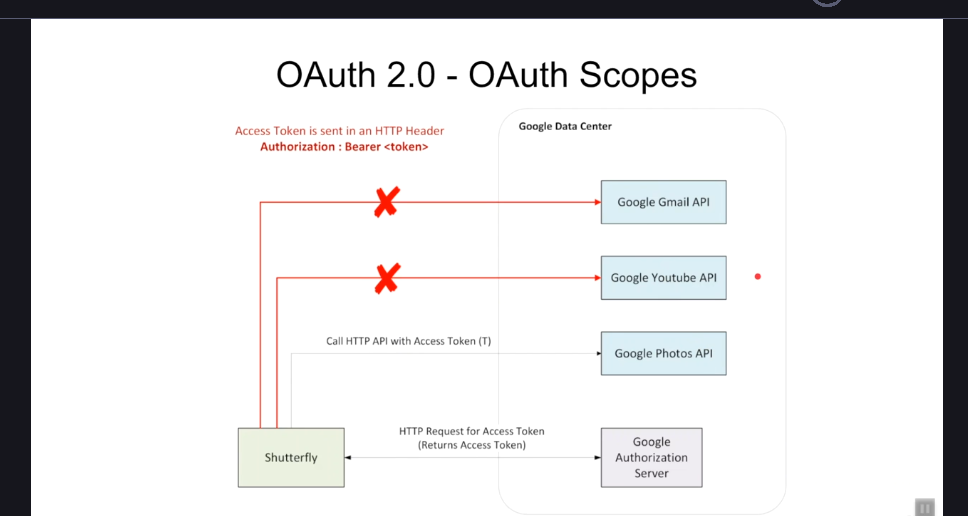
These are more **sensitive** scopes. Google might prompt for extra consent or require verification.



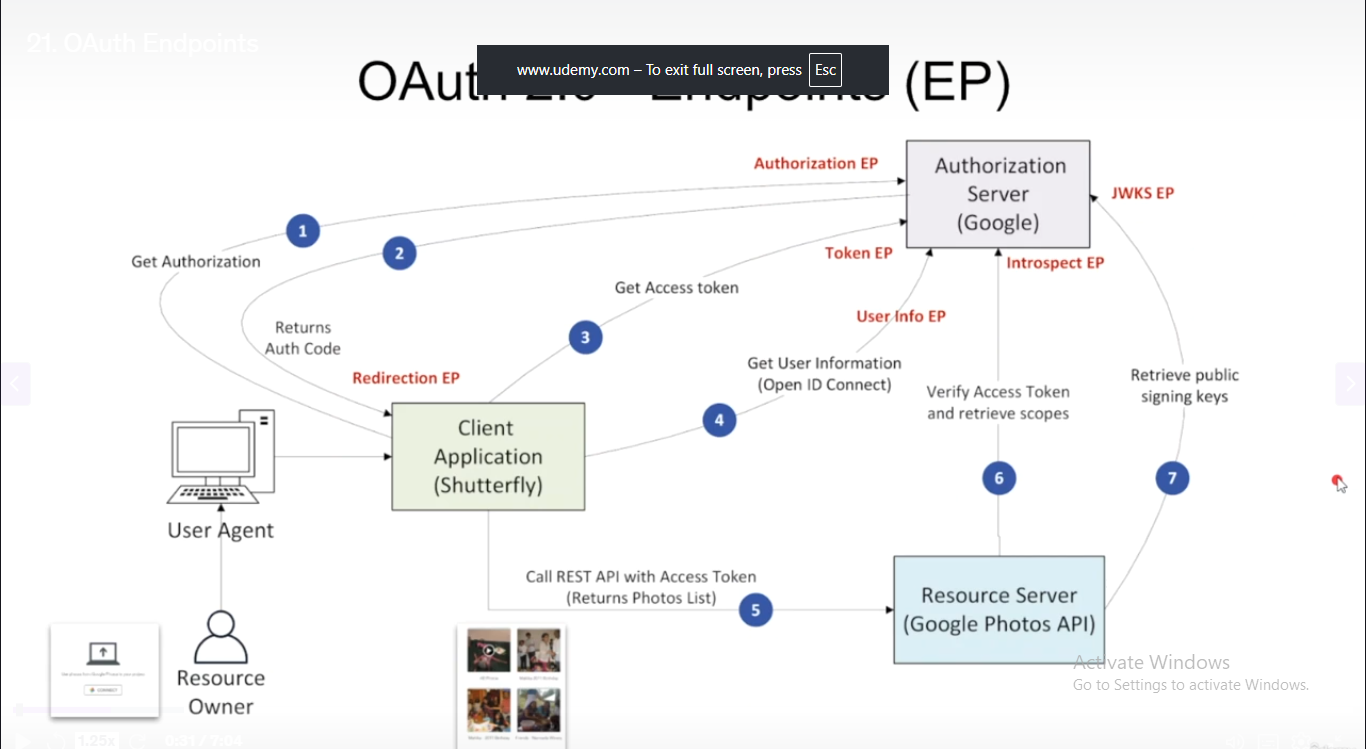
We can access any url based on scopes







If scope not available it wont allow



Why first we need to get oauth code and by using it gain we need to get token, why don’t we get token directly.?

Bcz oauth code sent in the browser from oauth server, so if anyone has access to the browser they can get it.

Where as the token endpoint called from the backend so it not saved in the browser.

So finally we do required two end points one to get the oauth code and second one is to get the token from the backend.

1. User visits: https://xyz-ramu.com/login

|

| (your frontend or backend redirects user's browser)

v

2. Browser redirects to:

https://accounts.google.com/o/oauth2/v2/auth?client\_id=...

|

| (user logs in and consents)

v

3. Google redirects back to:

https://xyz-ramu.com/oauth2/callback?code=abc123

|

| (your backend now handles it securely)

v

4. Your backend exchanges `code` for tokens

**Explain different grant types?**

Absolutely! OAuth 2.0 defines **four main types of grant flows**, also called **"grant types"**, and each is designed for a different kind of application or use case.

Let’s go through them **clearly**, with **real-world use cases**, diagrams (optional), and pros/cons.

**✅ 1. Authorization Code Grant (Most Secure & Common)**

**🔧 Use Case:**

Used by **web applications with backend servers** (e.g., Java Spring Boot, Node.js)

**🔐 Security:**

* Secure because tokens are exchanged server-to-server
* Client secret is **never exposed to browser**

**🔁 Flow:**

1. Redirect user to authorization URL
2. User logs in and approves
3. Authorization server sends **code** to your redirect\_uri
4. Your backend exchanges code + client\_secret for tokens

**✅ Example:**

Used in login via Google in Spring Boot app.

text

CopyEdit

Client (Browser) --> Google --> Redirects with code --> Backend exchanges code for token

**✅ 2. Client Credentials Grant**

**🔧 Use Case:**

Used by **machine-to-machine (M2M)** apps (no user involved)

**Example:**

* A Spring Boot microservice calling another service's API
* Backend script fetching service data

**🔁 Flow:**

1. App sends client\_id + client\_secret directly to /token
2. Gets access\_token

**🔐 No user login, just service authentication.**

text

CopyEdit

Backend --> Token endpoint --> Access token (no UI)

**✅ 3. Password Grant (Deprecated / Not recommended)**

**⚠️ Security Warning:**

This flow requires users to **share their username & password** directly with your app — bad practice!

**🔧 Use Case (historical):**

* Used in legacy apps
* Now **discouraged** in favor of Authorization Code + PKCE

**✅ 4. Implicit Grant (Deprecated)**

**🔧 Use Case:**

Used in **single-page apps (SPA)** like React, Angular (earlier)

**⚠️ Risk:**

* Tokens are returned in **URL fragments** (visible in browser)
* Cannot use client\_secret
* **No refresh tokens**

**🔁 Flow:**

1. Redirects user to Google
2. Google sends back access\_token directly

text

CopyEdit

Browser --> Google --> Returns access\_token in URL

➡️ Now replaced by Authorization Code + PKCE (see below)

**✅ 5. Authorization Code with PKCE (Modern Standard for SPAs & Mobile)**

**🔧 Use Case:**

* Used in **mobile apps**, **SPAs (React, Angular)** without backend
* No client\_secret needed

**🛡️ What is PKCE?**

* Proof Key for Code Exchange
* Prevents code hijacking in public clients

**🔁 Flow:**

Same as authorization code, but instead of client secret, app uses a **code\_verifier** and **code\_challenge**.

**🔄 Summary Table**

| **Grant Type** | **User Involved?** | **Secure?** | **Use Case** |
| --- | --- | --- | --- |
| Authorization Code | ✅ Yes | ✅✅✅ | Web apps with backend (Spring Boot, etc.) |
| Client Credentials | ❌ No | ✅✅✅ | Service-to-service (M2M) |
| Password (Deprecated) | ✅ Yes | ❌ | Legacy only |
| Implicit (Deprecated) | ✅ Yes | ❌ | Legacy SPAs |
| Auth Code + PKCE | ✅ Yes | ✅✅✅ | SPAs, mobile apps |

**✅ Real-Life Examples:**

| **App Type** | **Recommended Grant** |
| --- | --- |
| React app only | Auth Code + PKCE |
| Java Spring Boot with frontend | Auth Code |
| Backend API calling another API | Client Credentials |
| Legacy mobile app | Password (but migrate soon) |

sample end points where grant type passed?

###### ChatGPT said:

Absolutely! Let's look at **sample HTTP requests (endpoints)** where the grant\_type is passed for each type of OAuth2 flow.

These are always sent to the **token endpoint** of the OAuth provider (e.g., Google, Auth0, Keycloak, Okta, etc.).

### ✅ 1. ****Authorization Code Grant****

This is the most common grant (used with Spring Boot or Node.js backend).

#### 🔁 Token Exchange Request:

http

CopyEdit

POST https://oauth2.googleapis.com/token

Content-Type: application/x-www-form-urlencoded

grant\_type=authorization\_code

&code=abc123

&redirect\_uri=https://xyz-ramu.com/oauth2/callback

&client\_id=YOUR\_CLIENT\_ID

&client\_secret=YOUR\_CLIENT\_SECRET

### ✅ 2. ****Client Credentials Grant****

Used for **machine-to-machine** (M2M) communication, no user involved.

#### 🔁 Request:

http

CopyEdit

POST https://auth.example.com/oauth2/token

Content-Type: application/x-www-form-urlencoded

grant\_type=client\_credentials

&client\_id=YOUR\_CLIENT\_ID

&client\_secret=YOUR\_CLIENT\_SECRET

&scope=read:metrics

### ✅ 3. ****Authorization Code with PKCE****

Used in mobile apps and SPAs (no secret needed).

#### 🔁 Token Request:

http

CopyEdit

POST https://oauth2.googleapis.com/token

Content-Type: application/x-www-form-urlencoded

grant\_type=authorization\_code

&code=abc123

&redirect\_uri=https://your-spa.com/callback

&client\_id=YOUR\_CLIENT\_ID

&code\_verifier=your\_generated\_verifier

### ✅ 4. ****Resource Owner Password Credentials Grant**** (Deprecated)

#### 🔁 Token Request:

http

CopyEdit

POST https://auth.example.com/oauth2/token

Content-Type: application/x-www-form-urlencoded

grant\_type=password

&username=ramu@example.com

&password=ramuPassword123

&client\_id=YOUR\_CLIENT\_ID

&client\_secret=YOUR\_CLIENT\_SECRET

### ✅ 5. ****Refresh Token Grant****

Used to get a new access token without re-authentication.

#### 🔁 Token Refresh Request:

http

CopyEdit

POST https://oauth2.googleapis.com/token

Content-Type: application/x-www-form-urlencoded

grant\_type=refresh\_token

&refresh\_token=your\_refresh\_token

&client\_id=YOUR\_CLIENT\_ID

&client\_secret=YOUR\_CLIENT\_SECRET

## 🔍 Summary Table

| **Grant Type** | **grant\_type value** | **Where it's used** |
| --- | --- | --- |
| Authorization Code | authorization\_code | After user login |
| Client Credentials | client\_credentials | M2M service-to-service |
| Password (Deprecated) | password | Legacy login flows |
| Auth Code + PKCE | authorization\_code | SPA or mobile, uses code\_verifier |
| Refresh Token | refresh\_token | To renew tokens without re-login |
| so initially grant type grant\_type=authorization\_code and second time grant type =refresh\_token??  **ChatGPT said:**  ✅ **Exactly right!** You've got the flow perfectly. Here's a clear breakdown:  **🔐 Step-by-Step OAuth2 Grant Flow (Using Google, for example)**  **✅ First time login (initial authentication)**  ➡️ Your app uses:  grant\_type=authorization\_code  🔹 You send the **authorization code** (received after user login) 🔹 Google responds with:   * access\_token ✅ * refresh\_token ✅ * id\_token (optional, for OpenID Connect)   **🔄 Second time (or any time after token expires)**  ➡️ Your app uses:  grant\_type=refresh\_token  🔹 You send the **refresh token** 🔹 Google responds with:   * A new access\_token ✅ * (optionally) a new id\_token   🔹 You **don’t** need the user to log in again ✅ 🔹 No authorization code is required this time  **🔁 Flow Summary Table**   | **Scenario** | **Grant Type** | **What You Send** | **What You Get** | | --- | --- | --- | --- | | Initial login | authorization\_code | code, client\_id, client\_secret | access\_token, refresh\_token, id\_token | | After token expires | refresh\_token | refresh\_token, client\_id, client\_secret | New access\_token, optional id\_token | |  |  |

…………………………………………………………………………………………………………………………………..

**For below scenario explain What is resorce server and oauth server and cleint I have identity server and 10 micro services communicaing each other by using token?**

Great — let’s clarify your scenario by identifying the roles in OAuth2 terminology.

## ✅ ****Your Scenario****

You mentioned:

I have an **Identity Server**, and **10 microservices** communicating with each other using a **token**.

## 🧩 OAuth2 Roles in Your Architecture

| **Role** | **Who in your system** |
| --- | --- |
| **Client** | Web UI / Mobile App / External Service (that makes a request to your backend) |
| **OAuth Server** | Your **Identity Server** (e.g., Keycloak, Auth0, Okta, or custom Spring Authorization Server) |
| **Resource Server** | Your **10 microservices** — the services that **validate the token** and expose **protected APIs** |
| **Resource Owner** | The end-user who logs in (optional for machine-to-machine) |

………………………………………………………………………………………………………………………………………..

OPENID connect?

When you want get user information.

for this openId scope explain clearly with requests and use case?

Absolutely! Let's walk through a **realistic use case** of OpenID Connect with the openid scope — including:

* Step-by-step explanation
* Actual request examples
* Responses with id\_token
* What to do in your app with that info

**✅ Use Case: "Login with Google" for XYZ-Ramu App**

You want users to log into your website using their Google account (SSO).  
You're not asking for permission to access their Google Drive — you only need their **identity** (e.g., name, email, profile pic).

**🧩 Why OpenID Connect?**

Because you're authenticating the user, not accessing their data.

So you'll use:

ini

CopyEdit

scope=openid email profile

**🚀 Step-by-Step OpenID Connect Flow (with request examples)**

**🔵 Step 1: Your App Redirects User to Google**

📤 **Request to Google:**

http

CopyEdit

GET https://accounts.google.com/o/oauth2/v2/auth?

client\_id=XYZ\_CLIENT\_ID.apps.googleusercontent.com

&redirect\_uri=https://xyz-ramu.com/oauth2/callback

&response\_type=code

&scope=openid email profile

&state=abc123

✅ You are saying:

"Hey Google, please log in the user and send me an authorization code. I want their identity (openid), email, and profile."

**🟢 Step 2: User Logs In on Google and Grants Consent**

🔁 Google authenticates the user  
🔁 If user agrees, Google redirects back:

📥 **Redirect back to your app:**

bash

CopyEdit

https://xyz-ramu.com/oauth2/callback?code=AUTH\_CODE&state=abc123

**🟣 Step 3: Your Backend Exchanges Code for Tokens**

📤 **POST to token endpoint:**

http

CopyEdit

POST https://oauth2.googleapis.com/token

Content-Type: application/x-www-form-urlencoded

grant\_type=authorization\_code

&code=AUTH\_CODE

&redirect\_uri=https://xyz-ramu.com/oauth2/callback

&client\_id=XYZ\_CLIENT\_ID.apps.googleusercontent.com

&client\_secret=XYZ\_CLIENT\_SECRET

📥 **Response from Google:**

json

CopyEdit

{

"access\_token": "ya29.A0ARrdaM...",

"expires\_in": 3600,

"refresh\_token": "1//0gA...",

"id\_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...",

"token\_type": "Bearer"

}

**🟡 Step 4: Decode the id\_token (JWT) to Get User Info**

Use a JWT decoder (like [jwt.io](https://jwt.io)).

📥 **Decoded id\_token example:**

json

CopyEdit

{

"iss": "https://accounts.google.com",

"sub": "112233445566778899",

"name": "Ramu Bandela",

"email": "ramu@gmail.com",

"email\_verified": true,

"picture": "https://lh3.googleusercontent.com/a-/AOh14...",

"iat": 1711021234,

"exp": 1711024834

}

✅ Your app now knows:

* Who the user is (sub)
* Their verified email
* Their name and profile picture

**🟠 Step 5: Use this info to log them into your app**

In Spring Boot, you'd:

* Trust Google's JWT public keys to verify the id\_token
* Extract user info from id\_token
* Create a session or JWT for your app (if needed)

**✅ Summary Table**

| **Component** | **Value** |
| --- | --- |
| **Scope** | openid email profile |
| **Token received** | access\_token, refresh\_token, **id\_token** |
| **id\_token purpose** | Identity of the user |
| **Use case** | Logging in the user using their Google account |

**🌟 When to Use OpenID Connect (scope=openid)**

* Login via Google, GitHub, Okta, etc.
* SSO across multiple apps
* When you want to **authenticate**, not access protected data

…………………………………………………………………………………………………………………………………….

scope=openid email profile explain this?

Absolutely! Let’s break down this part of the OAuth/OpenID Connect request:

&scope=openid email profile

This tells the **OAuth Authorization Server** (like Google) *what kind of access or information your app is requesting from the user*.

**🔍 Breakdown of Each Scope**

**✅ 1. openid — (Mandatory for OpenID Connect)**

* This indicates you're **not just authorizing**, but also **authenticating** the user.
* When this is included, Google (or any OIDC provider) will return an **id\_token** — a JWT that contains the user's identity.
* Without openid, you won’t get id\_token, and you won’t be able to verify who the user is.

**What you get:**

* id\_token
* Contains claims like: sub (user ID), iss, aud, etc.

**✅ 2. email**

* Asks permission to access the user’s **email address**.
* Returned fields:
  + "email" (e.g., "ramu@example.com")
  + "email\_verified" (boolean)

**Example from decoded id\_token:**

"email": "ramu@example.com",

"email\_verified": true

**✅ 3. profile**

* Requests basic public **profile information**.
* Includes:
  + name
  + given\_name
  + family\_name
  + picture (profile pic URL)
  + locale

**Example from decoded id\_token:**

json

CopyEdit

"name": "Ramu Bandela",

"given\_name": "Ramu",

"family\_name": "Bandela",

"picture": "https://lh3.googleusercontent.com/photo.jpg",

"locale": "en"

**🧠 Why Use These Scopes Together?**

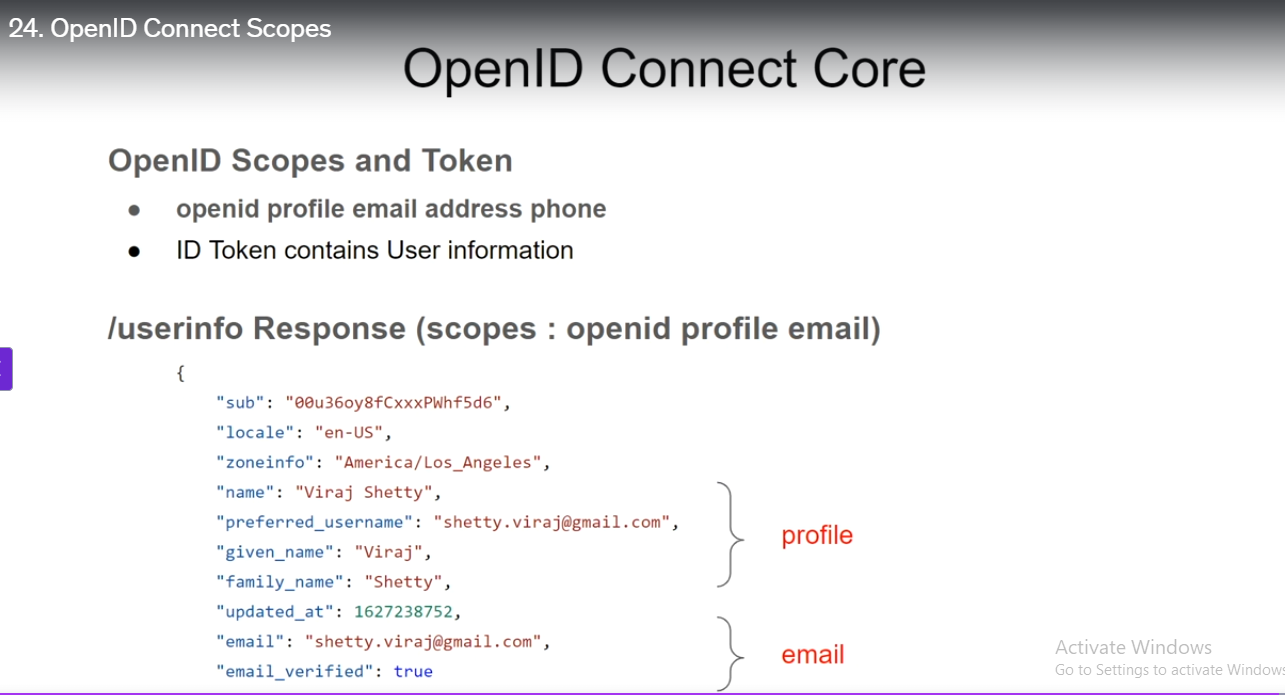
When you're building **"Login with Google"**, your app typically needs:

* openid → to authenticate the user
* email → to create an account or identify the user
* profile → to display user-friendly info (name, photo, etc.)

**📌 Summary Table**

| **Scope** | **Purpose** | **Sample Data Returned** |
| --- | --- | --- |
| openid | Required for OIDC and user ID token | sub, aud, iss, etc. |
| email | Email info and verification status | "email", "email\_verified" |
| profile | Public profile fields | "name", "picture", "locale" |

……………………………………………………………………………………………………………………………………………….



…………………………………………………………………………..

for openId any other scopes we can mention other than profile and email?

**ChatGPT said:**

Yes, absolutely! OpenID Connect (OIDC) defines a set of **standard scopes**, and you can also use **custom scopes** if supported by the identity provider (like Google, Okta, etc.).

**✅ Commonly Used OpenID Connect Scopes**

**1. openid (required)**

* Enables OpenID Connect (OIDC) login
* Returns an id\_token with identity claims

**2. email**

* Returns:
  + email
  + email\_verified

**3. profile**

Returns **basic profile information**, such as:

json

CopyEdit

{

"name": "Ramu Bandela",

"given\_name": "Ramu",

"family\_name": "Bandela",

"profile": "https://profile-url",

"picture": "https://photo.jpg",

"gender": "male",

"locale": "en",

"updated\_at": "2024-01-01T00:00:00Z"

}

**4. address**

Returns the user's **physical mailing address**:

json

CopyEdit

{

"address": {

"formatted": "123 Main St, City, Country"

}

}

**5. phone**

Returns:

json

CopyEdit

{

"phone\_number": "+91-1234567890",

"phone\_number\_verified": true

}

…………………………………………………………………………………………………………………………………………………………..

When multiple users accessing the micro services, on what basis the url access will be given in token

**ChatGPT said:**

Great question — and a very important one in securing microservices using OAuth2 or OpenID Connect.

When multiple users access microservices, **access to specific URLs (APIs)** is typically controlled using the **scopes** or **roles/authorities** embedded in the token (usually a JWT). Let me break it down step by step:

**✅ How URL Access Is Controlled Using Token**

**🔐 1. User Authenticates and Receives a Token**

* A user logs in via an **Identity Provider (IdP)** like Google, Keycloak, Okta, etc.
* The IdP issues an **access token** (and optionally an id\_token, refresh\_token).

**🧾 2. Token Contains Claims**

A typical JWT token will have fields like:

{

"sub": "1234567890",

"email": "ramu@example.com",

"roles": ["ADMIN", "USER"],

"scope": "read write profile",

"permissions": ["service-a:read", "service-b:write"],

"exp": 1711234567

}

**✅ 3. Each Microservice Verifies the Token**

Each microservice:

* **Decodes the token**
* **Verifies the signature**
* **Reads claims like roles, scope, permissions**

**🔁 4. URL Access Is Mapped Based on Claims**

**In Spring Boot (using Spring Security):**

You can configure access like:

http

.authorizeHttpRequests(auth -> auth

.requestMatchers("/admin/\*\*").hasRole("ADMIN")

.requestMatchers("/user/\*\*").hasAnyRole("USER", "ADMIN")

.requestMatchers("/service-a/read/\*\*").hasAuthority("service-a:read")

.anyRequest().authenticated()

);

This means:

* /admin/\*\* → only accessible if token has "roles": ["ADMIN"]
* /service-a/read/\*\* → accessible if "permissions": ["service-a:read"]

**🔄 Example Scenario**

| **User** | **Token Contains** | **Can Access** |
| --- | --- | --- |
| Alice (Admin) | roles: ["ADMIN"] | /admin/\*\*, /user/\*\* |
| Bob (User) | roles: ["USER"] | /user/\*\* only |
| DevOps Tool | scope: ["service-a:read"] | /service-a/read/\*\* |

…………………………………………………………………………………………………………………………………….

to get this whcih grant types and scopes will be used?

###### ChatGPT said:

To control **microservice URL access per user** using token claims like roles, permissions, or scope, here's exactly what grant types and scopes are typically used — broken down clearly:

## ✅ Step-by-Step Answer:

### 1️⃣ ****Grant Type to Use****

The most common and secure options for microservices:

| **Grant Type** | **When Used** | **Purpose** |
| --- | --- | --- |
| **Authorization Code** | End-user login flows via UI | Gets access token after login |
| **Client Credentials** | Machine-to-machine (e.g., one microservice calling another) | Gets token without user |
| **Refresh Token** | After login, for long-term sessions | Get new access token silently |

✅ For user-based UI login (frontend + backend):  
Use: \*\*authorization\_code\*\*  
✅ For service-to-service calls:  
Use: \*\*client\_credentials\*\*

### 2️⃣ ****Scopes to Use****

Scopes define **what kind of access** the token holder is requesting.

Typical scopes:

| **Scope** | **Purpose** |
| --- | --- |
| openid | Required for OpenID Connect (for identity) |
| profile | Name, picture, locale |
| email | Email address |
| read / write | General access (if API supports it) |
| service-a.read / service-b.write | Microservice-level fine-grained access |

### ✅ Realistic Example (User Login Flow)

#### 🔐 Request Token with:

http

CopyEdit

GET /authorize?

response\_type=code

&client\_id=xyz-client

&redirect\_uri=https://your-app.com/callback

&scope=openid profile email service-a.read service-b.write

Then exchange code for token via:

http

CopyEdit

POST /token

Content-Type: application/x-www-form-urlencoded

grant\_type=authorization\_code

&code=AUTH\_CODE

&redirect\_uri=https://your-app.com/callback

&client\_id=xyz-client

&client\_secret=xyz-secret

### 🧾 Sample Decoded JWT (Access Token):

json

CopyEdit

{

"sub": "123456",

"email": "ramu@example.com",

"roles": ["ADMIN"],

"scope": "openid profile email service-a.read",

"permissions": ["service-a:read", "service-b:write"],

"exp": 1712345678

}

### 🛠️ Microservice Access Control (Spring Boot)

java

CopyEdit

http

.authorizeHttpRequests(auth -> auth

.requestMatchers("/admin/\*\*").hasRole("ADMIN")

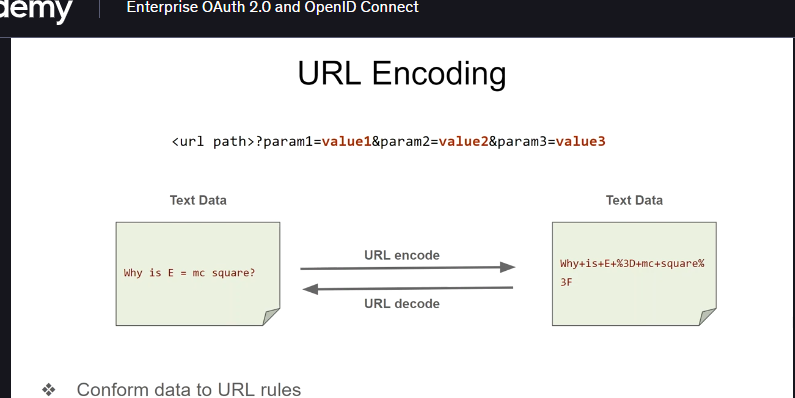
.requestMatchers("/service-a/read/\*\*").hasAuthority("service-a:read")

);

## 🧩 Summary Table

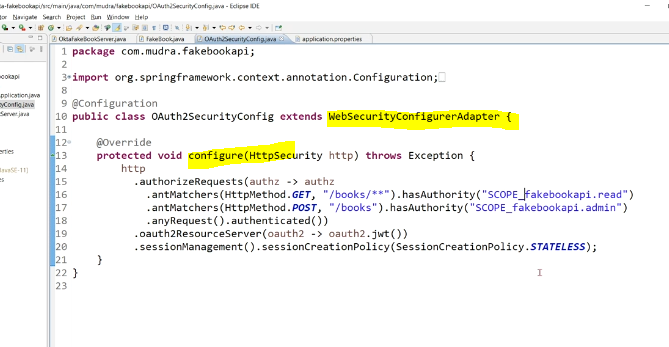
| **Scenario** | **Grant Type** | **Scopes** |
| --- | --- | --- |
| User logs in via UI (OAuth2) | authorization\_code | openid profile email + custom scopes |
| Microservice-to-microservice | client\_credentials | e.g., service-a.read |
| Long-term session continuation | refresh\_token | (no scopes; just refresh) |

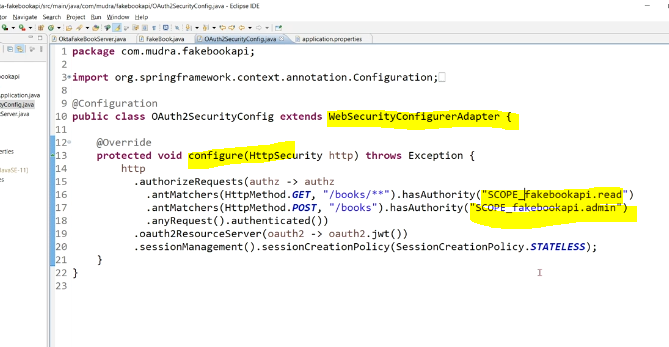
………………………………………………………………………………………………………………………………………..



…………………………………………………………………………………………………………………………………….

Example with Okta





@Configuration

public class OAuth2SecurityConfig {

// CHANGE : With Spring Boot 3.0, Create a SecurityFilterChain

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(authz -> authz

.requestMatchers(HttpMethod.GET, "/books/\*\*").hasAuthority("SCOPE\_fakebookapi.read")

.requestMatchers(HttpMethod.POST, "/books").hasAuthority("SCOPE\_fakebookapi.admin")

.anyRequest().authenticated())

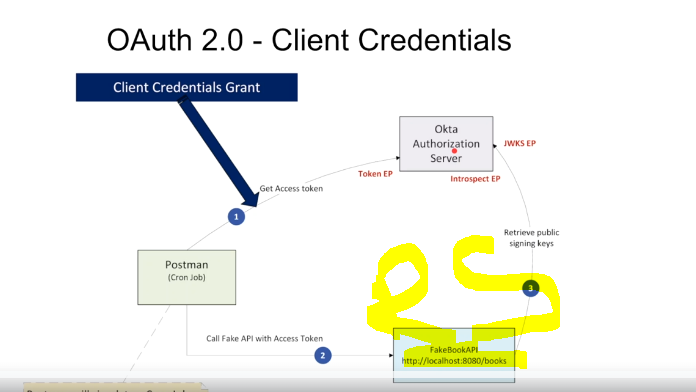
.oauth2ResourceServer(oauth2 -> oauth2.jwt(Customizer.withDefaults()))

.sessionManagement(sMgmt -> sMgmt.sessionCreationPolicy(SessionCreationPolicy.STATELESS));

return http.build();

}

}



**Actually we use client credentials grant type in micro services communication**

**We store the token in caffeine cache and we will get the new token when it is going to expire.**

## ✅ Best Practice in Production

| **Gateway/API Type** | **Strategy** |
| --- | --- |
| Public API Gateway | Use authorization\_code flow to authenticate user |
| Internal Services | Use client\_credentials to talk between trusted services |
| User-specific audit | Forward original token (if needed) |
| Secure fan-out (A → B → C) | Issue new tokens at each layer, or use token exchange |

Again if you sending token to the down streams by client credentails or auth code . micro services loose the single responsibiility

**ChatGPT said:**

Excellent insight — you're touching on a **core architectural principle**: **Single Responsibility**.

Your concern:

“If microservices are validating tokens (auth code or client credentials), aren't they violating the **Single Responsibility Principle (SRP)** by mixing business logic with security logic?”

**✅ You’re absolutely right to ask this.**

So let’s **break this down** carefully:

**🔍 What is the Single Responsibility Problem?**

Imagine a microservice that does this:

* 💼 Handles **business logic**
* 🔐 Validates JWT tokens
* 🔓 Checks user permissions/roles
* ❌ Now it’s doing **two responsibilities**:
  + Application behavior (domain logic)
  + Security enforcement (authz/authn)

That’s not aligned with **SRP**, and becomes harder to test, scale, and evolve.

**🧩 Solution: Separate Concerns Using a Gateway**

✅ This is why **API Gateways** and **Sidecars (like Istio Envoy)** exist.

| **Layer** | **Responsibility** |
| --- | --- |
| **Gateway** | Authentication & Authorization |
| **Microservices** | Domain/business logic only |

So ideally:

* ✅ The gateway **validates** the token (authn)
* ✅ The gateway **enforces** scopes, roles (authz)
* ✅ The gateway **routes** requests
* ✅ The token is either:
  + forwarded **as-is**, or
  + exchanged for a trusted **internal token**

And your microservices just assume:

“If I got this request, the user/service is already authenticated and authorized.”

**⚠️ When You Still Validate Tokens in Microservices**

Sometimes, microservices **must** validate tokens:

* When you **don’t fully trust** the gateway (e.g. multiple external entry points)
* When you need **per-service fine-grained authz**
* When you **forward user tokens end-to-end** (for RBAC, auditing)

**✅ Still SRP-safe if:**

* The auth logic is handled by a **library/middleware** (Spring Security)
* The service code just configures the rules — not doing manual parsing

**✅ Recommended Strategy for SRP-Friendly Auth Architecture**

**🛡 Gateway handles:**

* Token validation
* Scope enforcement
* Rate limiting
* Token exchange (if needed)

**🧠 Microservice handles:**

* Domain logic only
* Minimal @PreAuthorize("hasRole('ADMIN')") if absolutely needed
* Or better, trust identity and delegate deeper checks to shared libraries or external policies (e.g., OPA, AuthZ service)

**🧱 Layered Example:**

text

CopyEdit

[Frontend (user token)]

↓

[API Gateway]

- Validates token

- Checks permissions

- Routes to downstream

↓

[Microservice A]

- No token logic

- Pure business logic

**✅ Summary**

| **Topic** | **Best Practice** |
| --- | --- |
| Microservices doing security? | ❌ Avoid mixing unless necessary |
| Gateway responsibility | ✅ Full authn & authz enforcement |
| Token passed downstream? | ✅ Okay if trust boundary is strong |
| SRP compliant? | ✅ If auth logic is delegated or abstracted away |