OAuth2

What is OAuth2

Problems oauth solves

1. How different app of same organization can authenticated.(using oAuth this can be solve)
2. Delegated authentication and authorization
    a. Other app can access the drive

OAuth2

1. stand for Open Authorization
2. Open source and free open protocol for authorization

Grant Types

Flow of authorization

1. Authorization Code Grant

- **Description:** The Authorization Code Grant is the most secure and widely used grant type in OAuth 2.0 when an end user is involved. It's mainly used for web and mobile applications where the client cannot be fully trusted to store credentials securely.
- **Role of Authorization Code:**
    - The authorization code is a temporary code that the client receives from the authorization server after the user authenticates and consents. This code is then exchanged for an access token.
    - This flow ensures that the access token is not exposed to the user's browser, enhancing security.
    - Typically used with server-side applications.

2. PKCE (Proof Key for Code Exchange)

- **Description:** PKCE is an extension to the Authorization Code Grant type to make it more secure, especially for public clients like mobile and single-page applications that cannot securely store a client secret.
    - **Code Verifier:** A random string generated by the client and used only once.
    - **Code Challenge:** A hashed version of the code verifier. The client sends the code challenge during the initial authorization request.
    - **Process:**
        1. The client sends the code challenge to the authorization server during the initial request.
        2. When exchanging the authorization code for an access token, the client also sends the original code verifier.
        3. The authorization server verifies that the code challenge matches the code verifier to confirm the client's authenticity.

3. Client Credentials Grant

- **Description:** The Client Credentials Grant is used when the client (a machine-to-machine application) requests access to resources that are not owned by any specific end user.
- **Usage:** Common in microservices architecture where one service needs to communicate with another.
- **End User:** No end user is involved in this grant type; the client authenticates itself directly with the authorization server to obtain an access token.

4. Device Code Grant

- **Description:** This grant type is used for devices that do not have a browser or an easy way to enter credentials, such as smart TVs, IoT devices, or other hardware devices.
- **Usage:** The device displays a code to the user, who then enters it on another device (like a phone or computer) with a browser to authenticate. Once the user authorizes the device, the device obtains an access token.

5. Refresh Token

- **Description:** A Refresh Token is used to obtain a new access token when the current access token expires, without requiring the user to authenticate again.
- **Usage:** The refresh token is usually long-lived and can be exchanged for new access tokens, enhancing the user experience by reducing the need for frequent logins.

6. Implicit Flow (Deprecated)

- **Description:** The Implicit Flow is a simpler, but less secure grant type where the access token is issued directly to the client after the user is authenticated, without an intermediate authorization code.
- **Status:** Deprecated in OAuth 2.1 and not recommended for use in production because the access token is exposed in the URL, making it vulnerable to interception.
- **No Authorization Code:** The flow does not use an authorization code, which makes it less secure.

7. Password Grant (Deprecated)

- **Description:** In the Password Grant, the client takes the user's credentials (username and password) directly and sends them to the authorization server to obtain an access token.
- **Status:** Deprecated and not recommended for production use because it requires the client to handle and store user credentials directly, posing a significant security risk.

OAuth2 Terminology

Scenario: A Photo-Sharing Application

Imagine you have a photo-sharing application named **"PhotoHub"** that allows users to store, share, and manage their photos. Now, another application, **"CoolPhotoEditor,"** wants to access some of the user's photos on "PhotoHub" to offer photo editing features.

Here's how the different components fit into this scenario:

1. **Resource Owner (End User):**

- The **End User** is the person who owns the photos stored on "PhotoHub." This user wants to allow "CoolPhotoEditor" to access their photos for editing purposes but wants to maintain control over what the application can do.

2. **Client (Application that Reads the Resources):**
    - The **Client** in this case is **"CoolPhotoEditor,"** an external application that wants to access the user's photos on "PhotoHub" to provide its editing services.
    - The client must request access from the **Authorization Server** to get permission to access specific resources (photos) owned by the **Resource Owner**.
3. **Authorization Server (Server Managing Authorization Logic):**
    - The **Authorization Server** is a component of **"PhotoHub."** It handles the authorization logic and knows about the **Resource Owner** (the end user) and the permissions they have granted to various clients.
    - The **Authorization Server** authenticates the user and determines whether to grant access to "CoolPhotoEditor."
4. **Resource Server (Holds All the Resources):**
    - The **Resource Server** is another component of **"PhotoHub."** It holds all the resources, such as the user's photos.
    - Once the **Client** (CoolPhotoEditor) has been granted access by the **Authorization Server,** it will interact with the **Resource Server** to retrieve or manipulate the resources (photos).
5. **Scopes (Granular Permissions Client Wants to Access):**
    - **Scopes** define the specific permissions or access levels that the **Client** ("CoolPhotoEditor") is requesting.
    - For example, **"CoolPhotoEditor"** may request the following scopes:
    - `read_photos`: Permission to view the user's photos.
    - `edit_photos`: Permission to edit the user's photos.
    - The **Authorization Server** will display these scopes to the **Resource Owner** (end user) during the authorization process, and the user can choose whether to grant all, some, or none of the requested scopes.

OAuth Simple flow

1. Client register with auth server and get CLIENT_ID and CLIENT_SECRET
2. End user visited client , signup with auth server and user login auth server , user grant consent and now auth server issue token to client.
3. Now client access the resource server.

Example Flow

1. **User Initiates Authorization:**
    - The user (Resource Owner) opens **CoolPhotoEditor** and tries to use the feature that requires access to their photos on **PhotoHub**.
    - **CoolPhotoEditor** redirects the user to **PhotoHub's Authorization Server** to request permission.
2. **User Grants Permission:**
    - The user logs in to **PhotoHub** (if not already logged in) and is presented with a consent screen showing the scopes requested by **CoolPhotoEditor** (`read_photos`, `edit_photos`).
    - The user grants permission to access their photos for reading but denies permission to edit (`read_photos` scope granted, `edit_photos` scope denied).
3. **Authorization Server Issues Authorization Code:**
    - After the user consents, **PhotoHub's Authorization Server** redirects back to **CoolPhotoEditor** with an **Authorization Code**.
4. **Client Exchanges Authorization Code for Access Token:**
    - **CoolPhotoEditor** sends a request to **PhotoHub's Authorization Server** with the **Authorization Code** to obtain an **Access Token**.
    - The **Authorization Server** verifies the code and, if valid, issues an **Access Token** with the approved scope (`read_photos`).
5. **Client Accesses the Resource Server:**

- With the **Access Token, CoolPhotoEditor** makes a request to **PhotoHub's Resource Server** to access the user's photos.
- The **Resource Server** verifies the token and checks the scopes. Since only `read_photos` is allowed, it permits **CoolPhotoEditor** to read the photos but denies any attempt to edit.

6. **Access is Controlled Based on Scopes:**
   - **CoolPhotoEditor** can now access and display the user's photos but cannot make any changes to them since it only has the `read_photos` permission.

playground

https://oauth.net

https://www.oauth.com/playground/index.html

How the Resource Server Validates Tokens

When a client application (like "CoolPhotoEditor" in the previous example) sends a request to the **Resource Server** (e.g., "PhotoHub's" server that holds the photos), the **Resource Server** needs to validate the token (access token) provided by the client. There are two main approaches to validate tokens:

1. Validating Token Remotely

- **Description:** The **Resource Server** sends a request to the **Authorization Server** to check if the token is valid. This is a "remote" or "introspection" approach.
- **Steps:**
  1. When the client sends a request with the access token, the **Resource Server** forwards the token to the **Authorization Server** using an API call to validate it.
  2. The **Authorization Server** checks the token's validity (e.g., whether it's expired, revoked, or malformed) and responds with a status indicating whether the token is valid or not.
- **Pros:**

  - **Accuracy:** Always up-to-date since the **Authorization Server** is the source of truth.
  - **Revocation Handling:** Can handle token revocation scenarios effectively (i.e., when a token is manually revoked, it is immediately invalidated).
- **Cons:**
  - **Performance Overhead:** Requires an API call every time a token needs to be validated, which can introduce latency and reduce performance, especially in high-traffic situations.
  - **Network Dependency:** Requires the **Resource Server** to communicate over the network with the **Authorization Server**, making it dependent on network availability and the **Authorization Server's** responsiveness.

2. Validating Token Locally

- **Description:** The **Resource Server** validates the token locally without needing to call the **Authorization Server**. This is often done when using **JWT** (JSON Web Tokens) and involves cryptographic verification.
- **Details:**
  1. **Token Type - JWT:**
     - JWTs are self-contained tokens that include all necessary information about the token within the token itself (such as user information, scopes, and expiration time). They are digitally signed by the **Authorization Server**.
     - The **Resource Server** can decode the JWT and validate it locally without making an external call.
  2. **JWK (JSON Web Key) Approach:**
     - The **Authorization Server** signs the JWT using a private key.
     - The **Resource Server** downloads the public key (from a JWKS URL endpoint provided by the **Authorization Server**) to verify the digital signature of the token.
     - The **Resource Server** checks:
       - The **digital signature** using the public key to ensure the token has not been tampered with.

    - The **expiration time** of the token (embedded in the JWT) to ensure it is still valid.

OpenId

Protocol sits on the top of oauth2 for authentication.

Lets implement oauth2

InMemoeryClientRepository: register auth resources

CommonOAuthProvider : enum default server configurations

To configure auth server we can use both java and application properties

client_id

client_secret

Using KeyClock as Auth Server

1. Download and Install keyclick auth server to local machine
    a. download and install setup
    b. use docker for that.
2.

Kye hota hia

Open ID ?

Outh vs OpenId Connect ?

- Dono protocols milke kaam karte hain ek smooth user experience ke liye.

- **OpenID Connect** make sure karta hai ki aap kaun hain (authentication),
- aur **OAuth 2.0** allow karta hai ki woh app aapke data ko access kare (authorization), sirf aapke consent ke baad.