# Reference links

<https://oauth.net/>

<https://fusionauth.io/articles/oauth/modern-guide-to-oauth>

* [Client Credentials](https://www.oauth.com/oauth2-servers/access-tokens/client-credentials/) (oauth.com)
* [Client Authentication Methods](https://developer.okta.com/docs/reference/api/oidc/#refresh-token) (developer.okta.com)
* [Client Secret](https://www.rfc-editor.org/rfc/rfc6749#section-2.3.1) (RFC 6749 Section 2.3.1) – u can see all RFC here (Request for change)
* All about MTLS <https://datatracker.ietf.org/doc/html/rfc8705>

|  |  |
| --- | --- |
| video tutorials list | <https://developer.okta.com/blog/2022/06/16/oauth-java> |
| integration with java and java related frameworks | <https://oauth.net/code/java/> |
| spring urls | <https://github.com/spring-projects/spring-security/wiki>  <https://spring.io/projects/spring-security> |
| oauth playground | <https://developers.google.com/oauthplayground/> |
|  |  |

To learn PKCE <https://www.authlete.com/developers/pkce/>

<https://pazel.dev/teach-me-pkce-proof-key-for-code-exchange-in-5-minutes>

Open id

<https://openid.net/developers/how-connect-works/>

JWt

https://github.com/danvega/jwt-symmetric-key

Udemy course reference links

**Resource Server**

* **Repository(Original)**: <https://github.com/simplyi/ResourceServer>
* <https://github.com/simplyi/ResourceServer/tree/master-spring-boot-3.2.1-update>
* **Zip(Original)**: <https://github.com/simplyi/ResourceServer/archive/master.zip>
* **Repository(Updated to Spring Boot 3.2.1):**[Git Repository](https://github.com/simplyi/ResourceServer/tree/master-spring-boot-3.2.1-update)
* **Zip(Updated to Spring Boot 3.2.1):**[Zip](https://github.com/simplyi/ResourceServer/archive/refs/heads/master-spring-boot-3.2.1-update.zip)

**Albums Resource Server**

**Before configuring it as Eureka Client**

* **Repository:** <https://github.com/simplyi/Albums-Resource-Server/tree/Albums-Resource-Server-Before-Eureka-Client>
* **Zip:** <https://github.com/simplyi/Albums-Resource-Server/archive/Albums-Resource-Server-Before-Eureka-Client.zip>

**After Configuring Albums Resource Server as Eureka Client**

* **Repository:** <https://github.com/simplyi/Albums-Resource-Server/tree/Eureka-Client>
* **Zip:**<https://github.com/simplyi/Albums-Resource-Server/archive/Eureka-Client.zip>
* **Repository:**[Spring Boot 3.2.1 update](https://github.com/simplyi/Albums-Resource-Server/tree/Eureka-Client-Spring-Boot-3.2.1)

**Photos Resource Server**

**Before configuring it as Eureka Client**

* **Repository:** <https://github.com/simplyi/Photos-Resource-Server/tree/Photos-Resource-Server-Before-Eureka-Client>
* **Zip:**<https://github.com/simplyi/Photos-Resource-Server/archive/Photos-Resource-Server-Before-Eureka-Client.zip>

#### **Before**

**Photos Resource Server** before configuring it to be a Eureka Discovery Client

<https://github.com/simplyi/Photos-Resource-Server/tree/Photos-Resource-Server-Before-Eureka-Client>

**Albums Resource Server** before configuring it to be a Eureka Discovery Client

<https://github.com/simplyi/Albums-Resource-Server/tree/Albums-Resource-Server-Before-Eureka-Client>

**After configuring it as Resource Server**

* **Repository:** <https://github.com/simplyi/Photos-Resource-Server/tree/Eureka-Client>
* **Zip:** <https://github.com/simplyi/Photos-Resource-Server/archive/Eureka-Client.zip>

#### **After**

**Photos Resource Server** after configuring it to be a Eureka Discovery Client

<https://github.com/simplyi/Photos-Resource-Server/tree/Eureka-Client>

**Albums Resource Server** after configuring it to be a Eureka Discovery Client

<https://github.com/simplyi/Albums-Resource-Server/tree/Eureka-Client>

**API Gateway**

* **Repository:** <https://github.com/simplyi/ApiGateway>
* **Zip:** <https://github.com/simplyi/ApiGateway/archive/master.zip>

**API Gateway Before CORS Configuration**

* **Repository**: <https://github.com/simplyi/ApiGateway/tree/Before-CORSE-Configuration>
* **Zip:**<https://github.com/simplyi/ApiGateway/archive/Before-CORSE-Configuration.zip>

**API Gateway After CORS Configuration**

* **Repository:** <https://github.com/simplyi/ApiGateway/tree/CORS-Configuration>
* **Zip:**<https://github.com/simplyi/ApiGateway/archive/CORS-Configuration.zip>

**Discovery Service**

* **Repository:** <https://github.com/simplyi/DiscoveryService>
* **Zip:**<https://github.com/simplyi/DiscoveryService/archive/master.zip>

**Photo App Spring MVC Web Client Application**

* **Repository:** <https://github.com/simplyi/PhotoAppWebClient>
* **Zip:** <https://github.com/simplyi/PhotoAppWebClient/archive/master.zip>

**PKCE Example**

* **Repository:** <https://github.com/simplyi/PKCE>
* **Zip**: <https://github.com/simplyi/PKCE/archive/main.zip>

**Social Login/Logout**

* **Repository(Original):** <https://github.com/simplyi/SocialLoginWebClient>
* **Zip(Original):**<https://github.com/simplyi/SocialLoginWebClient/archive/master.zip>
* **Repository(Updated to Spring Boot v 3.1.2):**  [Repository](https://github.com/simplyi/SocialLoginWebClient/tree/master-update-to-Spring-3.1.2)
* **Zip(Updated to Spring Boot v 3.1.2):** [Zip](https://github.com/simplyi/SocialLoginWebClient/archive/refs/heads/master-update-to-Spring-3.1.2.zip)

**The** PKCE**-Enhanced Authorization flow in JavaScript application**

**JavaScript application**

* **Repository**: <https://github.com/simplyi/spa-example>
* **Zip:** <https://github.com/simplyi/spa-example/archive/main.zip>

**API Gateway**(before CORS configuration)

* **Repository:** <https://github.com/simplyi/ApiGateway/tree/Before-CORSE-Configuration>
* **Zip:** <https://github.com/simplyi/ApiGateway/archive/Before-CORSE-Configuration.zip>

**API Gateway**(after CORS configuration)

* **Repository:** <https://github.com/simplyi/ApiGateway/tree/CORS-Configuration>
* **Zip:** <https://github.com/simplyi/ApiGateway/archive/CORS-Configuration.zip>

**Resource Server**

**Resource Server**(before CORS configuration)

* **Repository:**<https://github.com/simplyi/ResourceServer/tree/CORS-configuration>
* **Zip:**<https://github.com/simplyi/ResourceServer/archive/CORS-configuration.zip>

**Resource Server**(after CORS configuration)

* **Repository(Original):**<https://github.com/simplyi/ResourceServer/tree/before-CORS-configuration>
* **Zip(Original):**<https://github.com/simplyi/ResourceServer/archive/before-CORS-configuration.zip>
* **Repository(Updated to Spring Boot 3.2.1):**[Git Repository](https://github.com/simplyi/ResourceServer/tree/master-spring-boot-3.2.1-update)
* **Zip(Updated to Spring Boot 3.2.1):**[Zip](https://github.com/simplyi/ResourceServer/archive/refs/heads/master-spring-boot-3.2.1-update.zip)

**Resource Server - Before adding Role-based access control**

<https://github.com/simplyi/ResourceServer/tree/before-adding-role-based-acceess-control>

**Resource Server - After adding Role-based access control**

<https://github.com/simplyi/ResourceServer/tree/after-adding-role-based-acceess-control>

**Resource Server - After adding Role-based access control(Spring Boot 3.2.1 Update)**

<https://github.com/simplyi/ResourceServer/blob/after-adding-role-based-acceess-control-spring-boot-3.2.1>

**Resource Server - Before adding method-level security code**

<https://github.com/simplyi/ResourceServer/tree/resourceserver-before-adding-method-level-security>

**Resource Server - After adding method-level security code**

<https://github.com/simplyi/ResourceServer/tree/resourceserver-after-adding-method-level-security>

**Resource Server - After adding method-level security code(Spring Boot 3.2.1 update)**

<https://github.com/simplyi/ResourceServer/blob/resourceserver-after-adding-method-level-security-spring-boot-3.2.1>

**Keycloak User Storage Provider SPI**

(Remote User Authentication)

**Remote User Storage Provider Implementation**

* **Repository**: <https://github.com/simplyi/KeycloakRemoteUserStorageProvider.git>
* **Zip:**<https://github.com/simplyi/KeycloakRemoteUserStorageProvider/archive/main.zip>

**Alternative to using RestEasy with Spring Boot 3.1+**

* **Repository**: <https://github.com/simplyi/KeycloakRemoteUserStorageProvider/tree/rest-easy-client-fix>
* **Zip**: <https://github.com/simplyi/KeycloakRemoteUserStorageProvider/archive/refs/heads/rest-easy-client-fix.zip>

**Legacy Users Web Service**

* **Repository**: <https://github.com/simplyi/LegacyUsersWebService.git>
* **Zip:**<https://github.com/simplyi/LegacyUsersWebService/archive/main.zip>

**The New Spring Authorization Server**

* **Repository:** <https://github.com/simplyi/NewSpringAuthorizationServer.git>
* **Zip:**<https://github.com/simplyi/NewSpringAuthorizationServer/archive/refs/heads/main.zip>
* [Authorization Code Grant Type](https://docs.google.com/presentation/d/1CiAiuay5rd1KDDnYwOyu6ud9xk5ZetSQDOMp9DYUKjs/edit?usp=sharing)
* [Client Credentials](https://docs.google.com/presentation/d/1KEA3i0F0bhB4me1uHfXkbmuaaFeRyxo7rG0ih-MlP68/edit?usp=sharing)
* [Password](https://docs.google.com/presentation/d/1kea9VCSP_QtQSb_NbU7MPOVLEF20iuOgieNW1g1MTwc/edit?usp=sharing)
* [Device flow](https://docs.google.com/presentation/d/1SlGr9z9bFIxYOLzwwZco3ny2W1XZ-9GJ9ORWA73gbjo/edit?usp=sharing)
* [PKCE-Enhanced Authorization code](https://docs.google.com/presentation/d/1yJeYPMoPY2050cZkkBcBu1SL5Z88StV7O7fH5f_6X3A/edit?usp=sharing)
* [Refreshing Access Token](https://docs.google.com/presentation/d/1e0bWzKk5JxxGXlAvAyeWe1q8iYgY804Y6vZ-zCpRAKU/edit?usp=sharing)

**A web client application - Initial empty project**

<https://github.com/simplyi/PhotoAppWebClient/tree/initial-project>

**A web client application - Example with RestTemplate**

<https://github.com/simplyi/PhotoAppWebClient/tree/rest-template-example>

**A web client application - Example using WebClient**

<https://github.com/simplyi/PhotoAppWebClient/tree/webclient-example>

OAuth – Open authorization, ok what is opened because of this?

Oauth2 started in 2012

What is OAuth and Why OAuth oauth flow?

Why oauth? With Oauth we don’t need to give our google app cred to another strange client ticket new app

OAuth, or open authorization, is a widely adopted authorization framework that allows you to consent to an application interacting with another on your behalf without having to reveal your password. It does this by providing access tokens to third-party services without exposing user credentials

The core OAuth 2.0 specification defines the "client password" (e.g. client secret) client authentication type, which defines the client\_secret parameter as well as the method of including the client secret in the HTTP Authorization header.

OAuth is a framework which enables clients to get the auth token from auth server

The end goal of Oauth is appln should get an access token from authorization server

With this we will follow Federated authentication mechanism, we don’t authenticate user, here authentication mechanism is delegated to auth servers like google, fb and app will receive auth token after authentication which we use that token by passing in every request

Oauth is a Federated authentication and delegated authorization which allows one app to access user data from other app on behalf of user ex:- ticketnew will fetch user details from google

Federated authentication in OAuth refers to a system where users authenticate with one trusted identity provider (IdP) to access **multiple services or applications** across different domains, using protocols like OAuth 2.0, SAML, or OpenID Connect.

|  |  |
| --- | --- |
|  | Oauth flow |

When to use OAuth

Use oAuth only when the user details are already present in one system , ex- book my show, linkedin when all these apps doesn’t have user data, since user data is already with google they are using oauth

Why it is called Open Authorization instead of authentication

Before oauth we should give our entire credentials to client 3rd party app, with that they can do whatever they want & access all Gmail, calendar, photos, drive mean unlimited scope

They came to solve this problem of limiting the access like only gmail or only drive, hence it is called open authorization instead of authentication

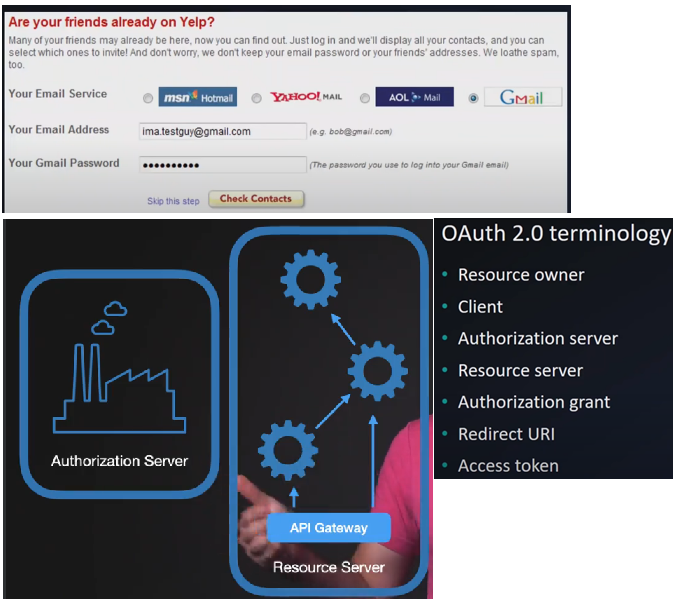
Before OAuth

Goal of OAuth- 3rd party apps also wants to access the data ,if ticket new app wants data from google, we should not give our google password to ticket new

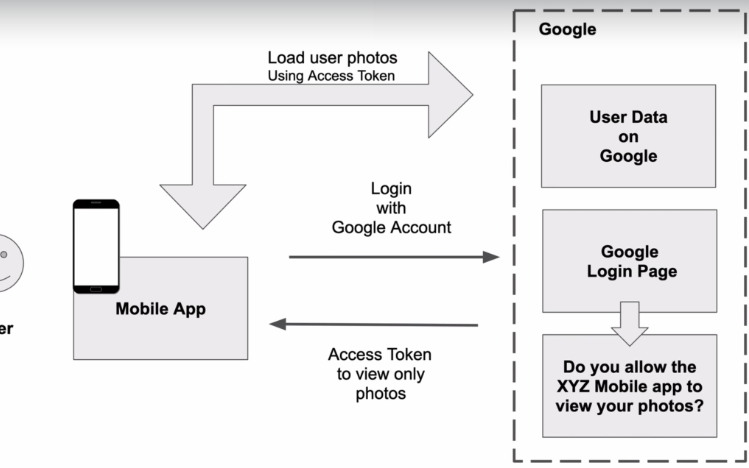
Before oauth we are supposed to give our credentials to random applications as below

In 2009 .. when 1 appn wants to talk to another application like if ticketnew app wants to get all user details that ticketnew application will take the **gmail user name and password** (which is very bad they can misuse the users Gmail id credentials ) and that application will talk to google and get all his information like name, age, mail id, .. ex:- apps book my show

So the main objective of oauth is not to share the password of google with ticketnew..



With oauth



## OAuth vs. open id connect

Open id means open identity

Open id connect is the identity layer built on top of OAuth

OAuth generates only token Top of Form

OAuth provides a way for an client application to get an access token to make API requests

In oauth2.0 u will get only auth token, but in open id connect u will get both auth token and ID token which contains user info like logged in user name, logged in user email id…These days every one is using open id connect only

Bottom of Form

OAuth is all about accessing api with token open id connect is about identifying the users, means OAuth

OAuth doesn’t provides the application with information about the user’s identity. That’s what OpenID Connect was created to add into the framework.

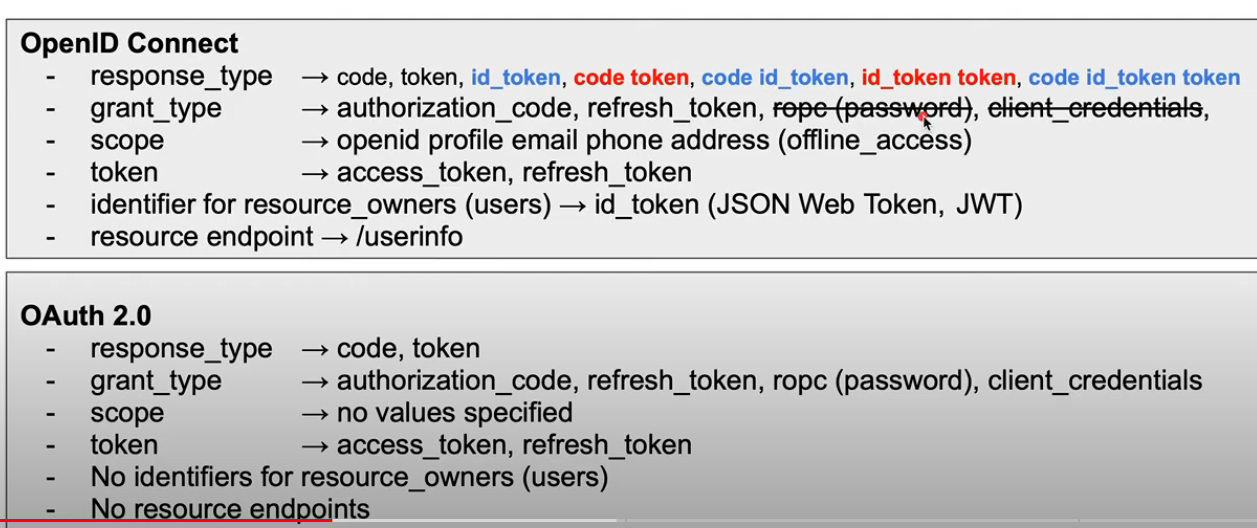
OIDC – open id connect is a simple identity layer on top of OAuth 2.0 protocol, it enables clients to verify the end user based on auth performed by auth server

* Use OAuth 2.0 if your primary concern is granting applications controlled access to user resources.
* Use OIDC if you need both authorization and user authentication (e.g., single sign-on) using existing login providers

**Authentication (verifying who you are):** **Entering a building with a keycard:**

**Authorization (verifying what you can do/ limiting access): even though u are employee of that building, some assoc they can’t enter other building as they are not authorized to enter**

|  |  |
| --- | --- |
| OAUTH | Open id Connect |
| Only for authorization just token,  no specific scope | For both authorization and authentication(restricted access)  It has the scope ex:- openid, with that token what and all can be accessed like email,photos |
|  | Open id connect has additional id token which has identity (user) info |
| No id (identity) token | Along with auth token we will get identity token which have some additional info about client/user |



My question?

with oauth we can limit the access means as we are already doing authorization, was it not authentication?

because a user will be authorized to enter any building after successful authentication right?

so do we still need to bother about authentication after successful authorization?

## OAuth 2.0 terminologies

**Client** –

Client is an application who is going to access user info on his behalf

it is nothing but a 3rd party application like ticket new app who wants to get our friends information from google oauth server

ex:- ticket new appln, this will get our details on our behalf

**Resource owner**:- who owns the data it is the person who has the data , ex:- These are the rooms that are accessible by the person who has the card

If you are building an API, using an external service as your Authorization Server, which roles are you building? –Ans resource server

Ex:- user

**Resource server/API**

It’s a place where our data is stored ex:- google photos api or our small micro service application

and we should hit this api with token which was issued by auth server,

When u hit the resource server with token, the resource server will take that token to auth server to validate if that token is correct or not, once auth server successfully validates that token then it will provide the response

**Authorization server** :- Is one who provides authorization token or access token to the client after validating our credentials& our ticket new api will use our token issued by auth server to access the google resource server api

The user will type the passwords only at Authorization server ex:- google auth server, **okta**, aws cognito, Microsoft identity platform, **keycloak**, spring authorization server,

Keycloak – is an auth server which can store user passwords its open source identity and access management server same like google auth server which can store passwords and issue token to clients

Supports SSO – means if u login to 1 app u will login to all apps, if u logout from 1 app then u will logged out from all apps

Supports social login, means using keycloak u can login with all google , facebook..

Here u can create client secret, scope,

**Analogy**: - when we go to hotel for stay they will verify our payment and they will give us a card, with that card we can access rooms, swimming pool, play area..

As like this card, we will receive the token from auth server once our credentials are verified, we can give that token to 3rd party applications like ticketnew and they will use that token and ask google for their contacts or photos

Ex:- if ticket new application wants to connect our google friends list, then unlike old way we will give our credentials to google oauth server he will verify our credentials(here google server is authorization server)

and give us the token we can give that token to ticket new application

Resource server:- where the resource is like google contacts api

Google contacts API is reusable secured API, if u want to use it authorization is mandatory, and

Similarly if u want to develop an reusable service ur api also should implement security and generally we get many requirements to consume an app on which oauth is enabled

**User agent /OAuth client** – the person or device who is making the request or

**THE API** – is the place where data lives or to whom we request

JWT / Auth token

|  |  |  |
| --- | --- | --- |
| <dependency>  <groupId>io.jsonwebtoken</groupId>  <artifactId>jjwt-api</artifactId>  <version>0.11.5</version>  </dependency> | <dependency>  <groupId>io.jsonwebtoken</groupId>  <artifactId>jjwt-impl</artifactId>  <version>0.11.5</version>  <scope>runtime</scope>  </dependency> | <dependency>  <groupId>io.jsonwebtoken</groupId>  <artifactId>jjwt-jackson</artifactId>  <version>0.11.5</version>  <scope>runtime</scope>  </dependency> |

Json Web token Ex:- From auth server we will get this auth token this a JWT token this contains so much of information,

There are 2 types of tokens, 1) By- value token – this contains lot of info in encoded format, which can be decoded easily this is called a JWT token

2) by reference token – this is just like a c++ pointer to the original data, it can’t be decoded, it doesn’t contain any info outside the network

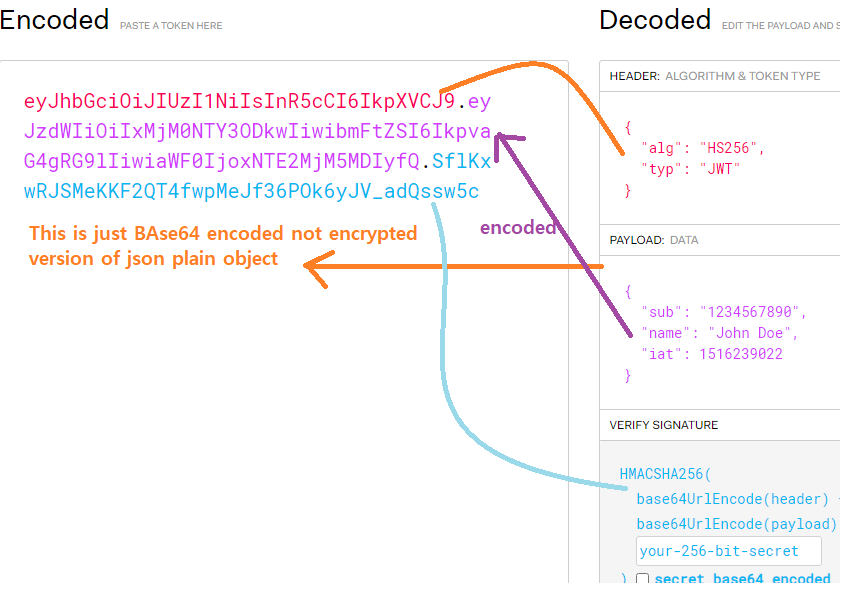
Only between trusted parties we will share JWT like ms-to micro service communication, if u are travelling outside of network then use by reference token

Note:- since the JWT is just an encoded form, jwt payload can be Base64 decoded very easily without any key, it should never contain any sensitive info like credit card

|  |  |
| --- | --- |
| JWT advantages | Dis advantages |
| 1) self contained infor, portable, and don’t require server side storage |  |

1. **Signing Process:**
   * The JWT is composed of three parts: header, payload, and signature.
   * The header specifies the signing algorithm used (e.g., HS256, RS256).
   * The payload contains the claims, which are pieces of information about the subject (user, application, etc.).
   * **A secret key (known only to the issuer) is used along with the header and payload to generate a digital signature.**
   * This signature is then appended to the JWT, forming the final token.
2. **Verification Process:**
   * When the receiver gets the JWT, they separate the header, payload, and signature.
   * They use the same signing algorithm specified in the header.
   * They use their own public key (if public key cryptography is used) or the issuer's public key (if available) to recreate the signature based on the header and payload.
   * They compare the recreated signature with the one included in the JWT.
3. **Tampering Detection:**
   * If the recreated signature matches the received signature, it indicates that the JWT has not been tampered with.
   * Any change in the header, payload, or even a minor character alteration would result in a mismatch between the recreated and received signatures. This mismatch exposes tampering and the token is rejected.

How to generate a JWT- Take a plain Json object (with header and paylaod) and Base64 encode that & then u will get that token



Sample Original payload

"exp": 1709714041, //expires at

"iat": 1709713741, // issued at

"auth\_time": 1709711525, //Time when authentication occurred

"jti": "c0b213b1-7161-43d9-9fb5-14c55fd335d0", unique identifier for this token

"iss": "http://localhost:8080/realms/hydrealm", //issuer who created this token

"aud": "account", // who requested it /audience ex:- client

"sub": "24846346-db03-439f-9f5b-0864e89010ef", // it is for user for whom JWT was issued or to whom we should give the token// whom the token belongs to // this is

"typ": "Bearer", //type of token

"azp": "1dstr-wellsfargo", // authorized party means who is authorized // the party to whom the token was issued

"sid": "21b282ff-3106-4129-9bc6-81ea8288ec87", //session id

* Imagine a JWT as a **letter of recommendation** for a user (subject) applying for a job (accessing an API).

The subject () is the person the letter is written about, highlighting their skills and experience

User = who will get the benefit/ whose info does the JWT have, ex:- the letter have person info, similarly the token have user info not client app info

* The authorized party (who is going to process his benefits) is the company (API) receiving the letter and considering the user's application.

subject:- It is the person who is going to get benefit

* The subject doesn't necessarily have to be a user. It could be another service or application involved in the authorization process. However, in most common scenarios, it's the user.
* The authorized party can be a single application or multiple applications that trust the issuer of the JWT and recognize the subject's identity.
* It's possible, though less frequent, for the subject and authorized party to be the same entity. For instance, a JWT used for internal communication within a microservices architecture might have the same service as both the subject (claiming its identity) and the authorized party (receiving the JWT for further processing).

How to decode JWT

Just Base64 Decode it

String encodedToken = "SGVsbG8gV29ybGQ=";

// Decode the token

byte[] decodedBytes = Base64.getDecoder().decode(encodedToken);

String decodedToken = new String(decodedBytes);

// Print the decoded token

System.out.println("Decoded token: " + decodedToken);

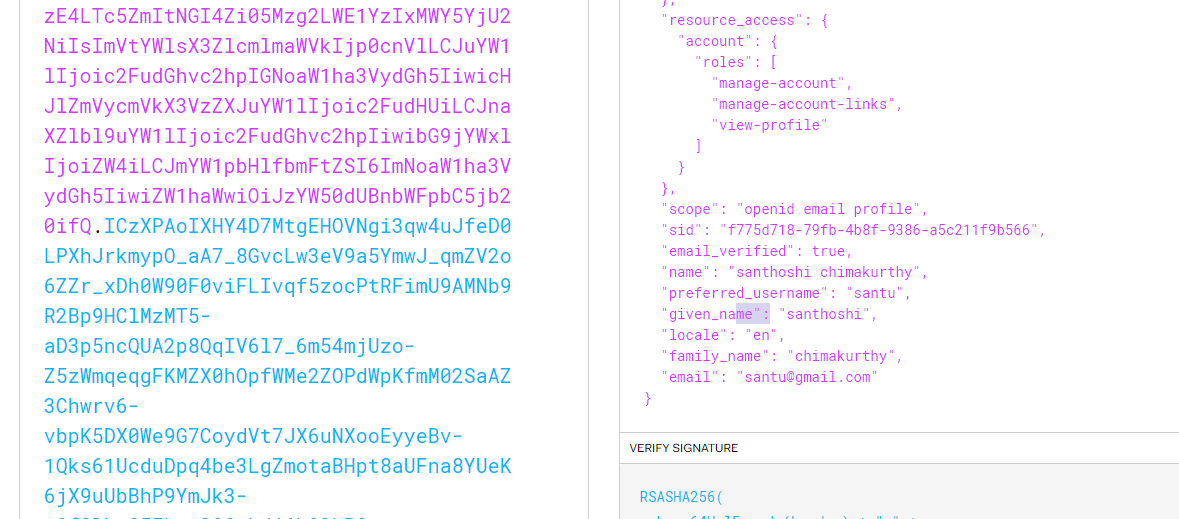
Is this JWT secure??

Since it is just encoded without any key, anybody can catch and decode that token without any key and hence any one can see the original data

Pending- learn more from youtube

Sample JWT token which can be decoded by jwt.io website

eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI5SndUUUdxb1E5QlVwLWdiNUt2LU9zTnQ4cGlON0cxc0FYRllqX2stZklzIn0..ICzXPAoIXHY4D7MtgEHOVNgi3qw4uJfeD0LPXhJrkmypO\_aA7\_8GvcLw3eV9a5YmwJ\_qmZV2o6ZZr\_xDh0W90F0viFLIvqf5zocPtRFimU9AMNb9R2Bp9HClMzMT5-aD3p5ncQUA2p8QqIV6l7\_6m54mjUzo-Z5zWmqeqgFKMZX0hOpfWMe2ZOPdWpKfmM02SaAZ3Chwrv6-vbpK5DX0We9G7CoydVt7JX6uNXooEyyeBv-1Qks61UcduDpq4be3LgZmotaBHpt8aUFna8YUeK6jX9uUbBhP9YmJk3-uQf0DknG5Fhxr3QCrLJj6hO2kP8o-HGn22SbFkFh-ygjuHQ

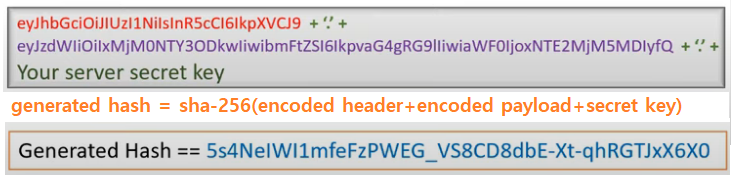


With below code u can simply decode the encoded token See above is a JWT token which we can easily extract at jwt.io



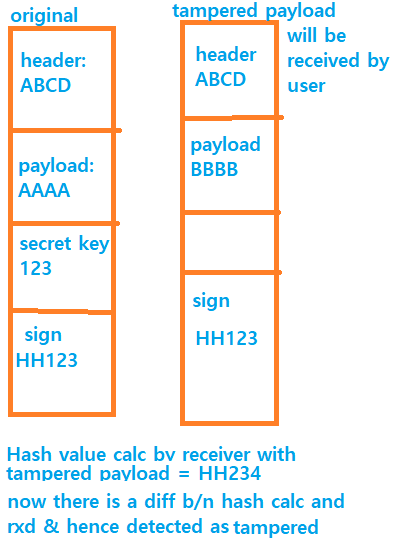
Is this secure?- I think no because signature =sha-256(Base64(header)+ base64(payload) )

Now what I will do, I will change header and I will change content of payload and I will generate signature accordingly by hashing or sha-256 now even if they check signature it will be equal to that



|  |  |  |
| --- | --- | --- |
| Header | Payload | Signature must be equal to sha-256(Base64(header)+Base 64(payload)+**secret key**) |
| This is the original data {abcd} | {name:”Mani”} | Now I will generate signature as above with secret key |
| This is the tampered by hacker{abce} | {name:”Gani”} | **Symmetric key pair:- Same key is used to sign the token and same key is used to verify the sign**  Creating tampered payload🡪 Now as the payload is just base64 encoded he can easily decode and changes the content and keeps tampered encoded payload, hacker can’t generate the signature as to generate the signature with new payload he needs the public key or private key ,  Original sign= SHA-256(secret key+ Base64(header) +Base64(payload)) = ZZAZA  **since he is not having that secret key he can’t generate and he will use old sign as it is**  **Regenerating the signature :-** At the end receiver re-generate the signature with tampered payload and secret key which will be diff with the original sign and rx will understand that sign or payload is tampered and he will rejects the JWT  Sign calculated by receiver= SHA-256(secret key+ Base64(header) +Base64(**Tampered payload**)) = HaAZA  Therefore the signature which is calculated by receiver and sign present in JWT differs and receiver will suspect the change and reject the JWT    **Asymmetric key pair**  In asymmetric key pair one key is used to sign the token and another key is used to verify the signature |

Tip:- if the id token is received from a SSL u can skip signature validation



Since this is only encoded, sensitive info should not be sent using JWT payload

Signing just ensures the data is not tampered I don’t know how? Signing can be done in 2 algorithms, symmetric algorithms and asymmetric algorithms

Symmetric algorithms (same key):- HMAC HS256 uses a same key to sign and verify, these are quick and simple

Asymmetric algorithms (different key):- uses a private key to sign the token and a public key to validate the token

JWT claim

It is like metadata – means data about data = it is like some main important data about the subject (whose info is there in the token)

JWT claims are the essential data packets carried within a JSON Web Token (JWT).

These claim reveals details about the subject (typically, the user) and other relevant data critical for authorization

This is a sample registered claim

{

"exp": 1710150831,

"iss": "http://localhost:8080/realms/hydrealm",

"aud": "account",

"sub": "24846346-db03-439f-9f5b-0864e89010ef",

}

1. **Payload:** Contains the JWT claims in JSON format

Types of claims: - 1) Registered claims 2) public claim 3) Private claim

* **Registered Claims:** These are a predefined set of standardized claim names outlined in the official JWT specification (RFC 7519). They are optional but highly recommended to ensure interoperability between different systems that understand JWTs. Some common examples include:
  + iss (issuer): Identifies the party that created and signed the JWT, like an authorization server.
  + sub (subject): The entity associated with the JWT or whose info is present in JWT, often the user identity.
  + aud (audience): Intended recipient(s) of the JWT like who is going to process the JWT like user will give to whom, typically a specific application or API.
  + exp (expiration time): The timestamp indicating when the JWT becomes invalid.
  + **Public Claims:** These are custom claims defined at will by the parties involved in the JWT exchange. There's more flexibility here. You can include any information relevant to the authorization process as long as both the issuer and receiver understand the meaning. Examples include user roles, permissions, or even specific data related to the user (like their preferred language).
  + **Private Claims:** These are custom claims intended only for the parties directly involved (issuer and recipient). They might contain sensitive information and are not necessarily parsable by third-party systems because they are not part of the JWT standard. An example could be an internal user identifier within a company's authentication system.

Client Secret- it is the password which will be given to the app after registration, it will be given only to confidential clients who can maintain passwords

Public clients: These applications are not capable of maintaining those client secrets like passwords/secret key and these apps are incapable they won’t/shouldn’t maintain any client secrets

If u are writing single page application ur app should not maintain any secret, if u maintain, user can hack that client secret key which will be misused

Note: [PKCE](https://oauth.net/2/pkce/) is not a form of client authentication, and is not an alternative to client authentication. Applications using client authentication should also use PKCE.

Ex:- Mobile apps, single page browser based app don't have the ability to be deployed with a client secret, since users of the app would be able to extract the secret. Therefore they are considered public clients.

Mobile apps don’t have the ability to be deployed with credentials, so they are considered public clients. The users of the app are not significant to this distinction.

Confidential clients: Confidential clients are those clients which are capable to maintain the client secret,

So these clients while registering themselves under an auth server, they register as confidential clients

These clients have some passwords like Client secret/private key JWT / MTLS or these client like java applications can maintain the client secrets

Confidential clients authenticate when making requests to the OAuth authorization server.

For first party confidential client authorization server may skip the consent screen because only trusted client can hit the application with the client secret

For confidential clients consent screen is skipped

Ex:- java, .net application can maintain client secrets effectively

Front Channel

Also called implicit flow

Sending data in url like the way which is visible

Using the browser’s address bar to move data between two other pieces of software is using the front channel.

Ex:- we should never send the oauth token in front channel (url )

window.location = 'https://authorization-server.com/authorize?client\_id=example';

here in this way data is visible in url here this is front channel and it is less secured way

Back Channel

Sending the data/oauth token through a program or through headers not urls &

it is possible even in java script also

For the purposes of OAuth, back channel means a secure HTTP connection from any HTTP client to a server Any HTTP client that makes a request to an HTTP server is using the back channel, even if that client is JavaScript code in a browser

Back channel is always secured

fetch("https://authorization-server.com/", {

method: "GET",

headers: {

"Content-Type": "application/json"

}

})

.then((response) => {

return response.json();

});

Here we are sending data in headers hence this is not visible in url so this is called back channel

Client Id

Every appln is uniquely identified by its client id, once we register we will get the client id from authorization server, It is the public information

While authorization we should send our client id, based on that only it will show that particular app’s consent screen to the user

SHA- 256

SHA- means secure hashing algorithm, Hashing is a process of scrambling original data the diff between hashing and encoding is hashed value is irreversible can be broken back to original string

SHA-256, SHA-512 output hashed values will be irreversible, no hacker can crack the hashed value, its impossible to get original value once u hash

Ex:- when users login into a website, we should not store direct plain user name & passwords, we should hash the password and user name and store it, next time if same user came and enters same pass we will verify the hash value

So it is always mandatory to store only the hashed passwords instead of plain passwords

means secure hashing algorithms used to encode or provide a 256 bit hash output

**App registration**

While registration we should give our redirect url so that auth server after authentication they will redirect to our specified url with the auth token

On successful registration we will get our client id (which will be unique to identify any client) and client secret (password)

KeyCloak server setup

First, keycloak is an authorization server, Go to <https://www.keycloak.org/downloads> zip sha 1

To start keycloak server go to bin directory and type “kc.bat start-dev” to start in development mode

By default, master realm will be there, here realm means space, create a new realm of any name

Create users only under diff realm

### What is MTLS

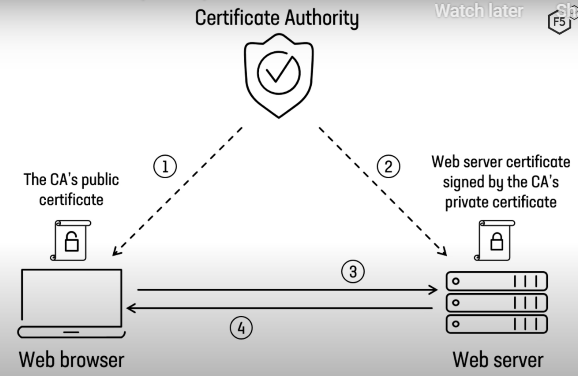
SSL, TLS both are technology used to encrypt the data communication

MTLs- mutual TLS means both server and client will verify other’s certificate to confirm whether they are talking to right applications or not

So in postman when u are hitting an MTLS URL u need to attach certificate (so there postman client will verify server certificate, server will postman s certificate, hence we need to attach certificate)

Because both server and client needs to verify others certificate

MTLS (Mutual transport layer security) which is a technology establishes an encrypted connection in which both parties use x509 digital certificates to authenticate and verify each other



Forms of client authentication

Here client means application who wants the resource

These are most common forms of client authentication.

* [Client Secret](https://www.rfc-editor.org/rfc/rfc6749#section-2.3.1) (RFC 6749 Section 2.3.1)
* [Mutual TLS](https://oauth.net/2/mtls/) (RFC 8705) [MTLS](https://tools.ietf.org/html/rfc8705) is a form of [client authentication](https://oauth.net/2/client-authentication) and an extension of OAuth 2.0
* [Private Key JWT](https://oauth.net/private-key-jwt/) (RFC 7521, RFC 7523, OpenID)

1. Using client secret-> Clients Applications sending the client secret to authorization server is not more secure practice (this is most common as this is easy to use as this is even static key),

Ex: Client should send the secret in header as below

Authorization: Basic czZCaGRSa3F0Mzo3RmpmcDBaQnIxS3REUmJuZlZkbUl3

Or

lient\_id

REQUIRED. The client identifier issued to the client during

the registration process described by [Section 2.2](https://www.rfc-editor.org/rfc/rfc6749#section-2.2).

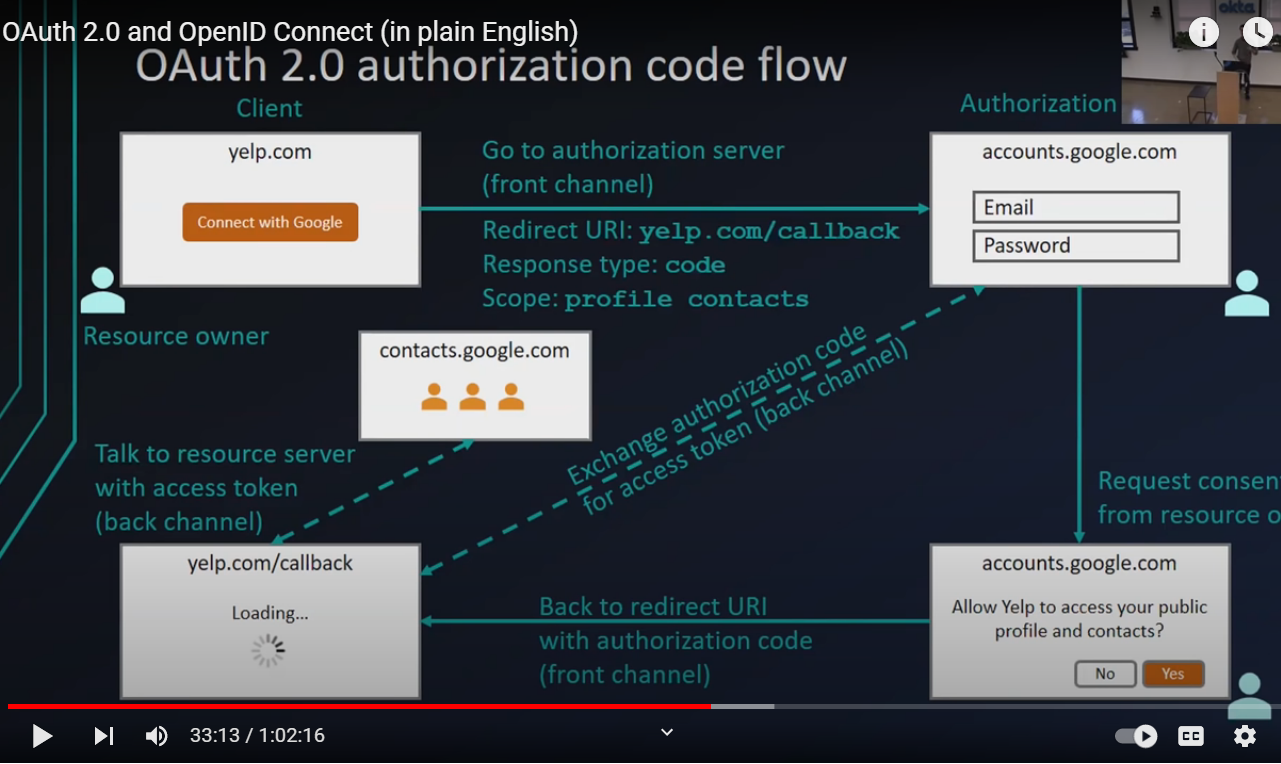
client\_secret

REQUIRED. The client secret. The client MAY omit the

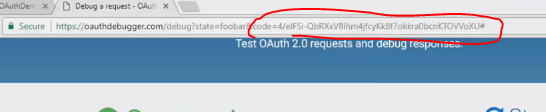
parameter if the client secret is an empty string.

These will be passed as the headers & authorization servers are responsible to decode the key

1. You should use the private key and public key concepts using Mutual TLS (MTLS) or a private key to sign a JWT



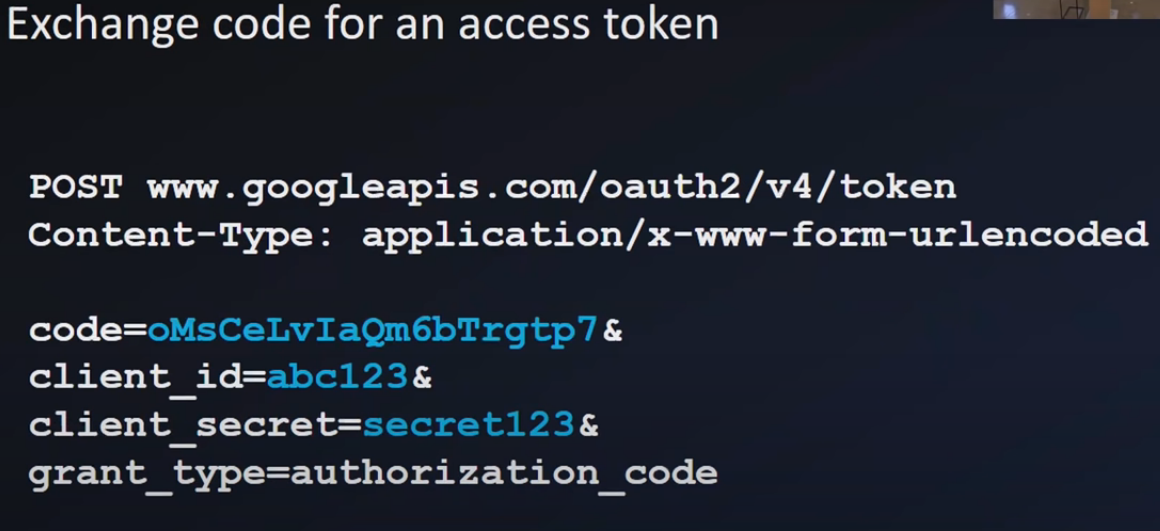
When user provides consent we will get authorization code in the browser address bar, not access token, we will hit again with auth code to get access token and with that we will hit resource server to get contacts



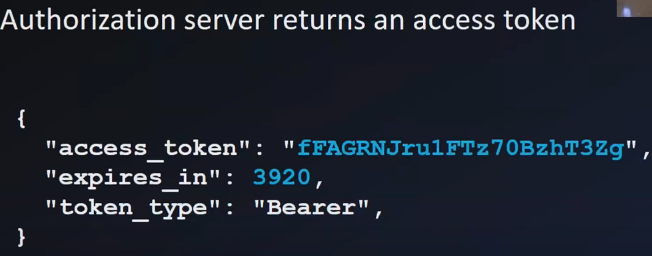
## Procedure to get the access token

* 1. Once user provided his credentials on consent screen, then after authentication oauth server will send auth code in front channel (url/address bar which is not confidential)
  2. With
  3. )auth code and
  4. unique client identifier(for each client) and
  5. with his client secret &
  6. pkce identifier/unique hash code for that flow

With all these details hit the auth server again and get the auth token in back channel (secret information in headers)



Once u got the auth code from the url address bar and hit google api from backend and get the access token,



It has expiration time, as this is valid only for some time and this is valid only for some url like whatever the user have gave the consent if user gave consent to read access to contacts, he can only access them he can’t delete their contacts

When u are completely using front end – react alone or javascript alone then u will directly get the access token in the url I think , because with javascript we may not be able to call backend server to get the access token from the auth code\

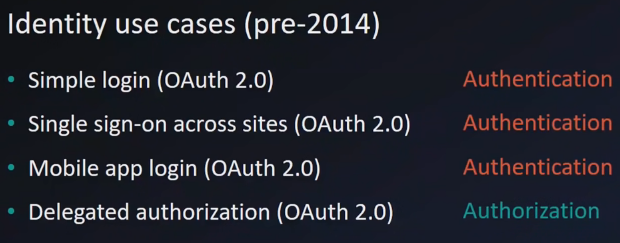
## Use cases of OAUTH

Now auth is used for both authentication and authorization

Like login with facebook , means

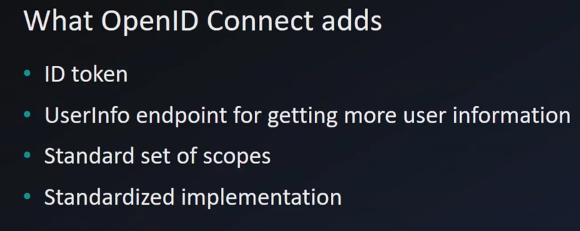
Authorization use case:- to use face book contacts first login and provide the consent and allow access to use contacts

Oauth wasn’t designed to use for authentication, its was designed for authorization only



Now these days they are using open ID connect for authentication and oauth for authorization

OPEN Id connect is just an extension to oauth



If I am talking to an oauth server which can understands open ID connect

I will get access token to talk to resource server and ID token ,,but what s the use of this

And use this access token and hit user info end point and get more details about the use for authorization

Sample O-Auth flow

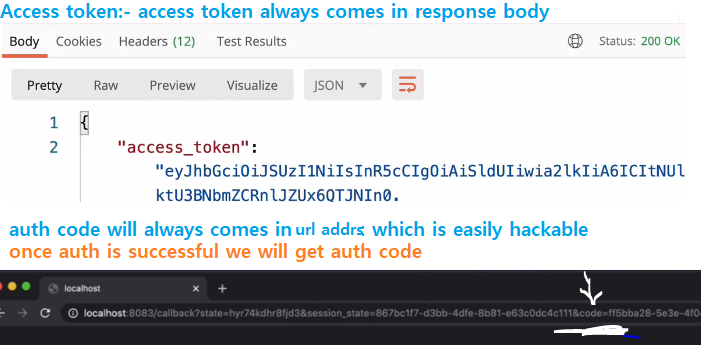
Client- ticket new application

When user provides credentials to auth server, Auth server issues a token to client appln and client uses this token to communicate with resource server and resource server will validate this token with auth server and once auth server confirms then requested data will be served to client

Its totally 2 steps

* 1. enter credentials🡪 get auth code, Request to get auth code from auth server 🡪 **hit url ,redirect to login page, enter credentials, auth server will validate, if success we will get auth code**
  2. Pass auth code, client secret, client id & get auth token from auth server

Auth code vs auth token



As auth code comes in url address bar, a fake appln can also have same app url as ur and steal auth code ,hence concept of PKCE came

Why not token directly? Why auth code again?

Since UI Front channel is there we are supposed to go with 2 step process, if java ms- to- java ms means in single step we will get token directly instead of auth code

Front channel means address bar is not secure anyone can tap easily, in step-1 auth code will be sent to us in address bar url, incase if u want single step with out auth code if u want auth token directly

1)**If auth token comes in address bar then anyone can see easily using dev tools or in url itself and anyone can easily Base64 decode that encoded one so it needs to be sent through back channel only**

**As front** channel request is mandatory as client should login, that’s why since front channel is less secure 1st auth code comes

In 2nd request as we need it securely we will get auth token in back channel ex:- headers or REST response

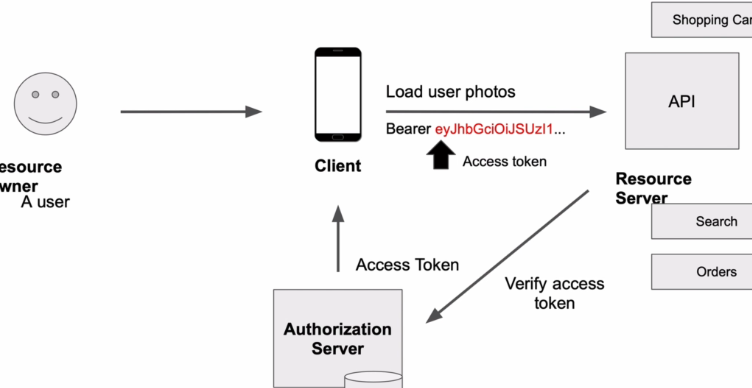
Therefore since we are using front channel (for user auth to enter cred) and if token Is passed directly in header it will be easily theft and seen

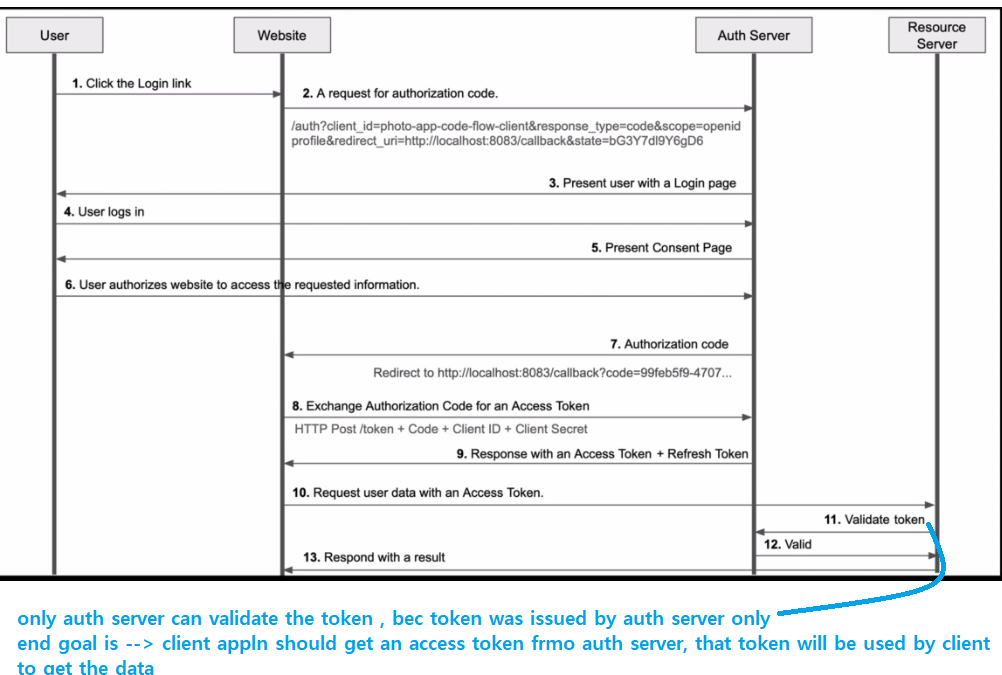
As token needs to be sent in secure way again a back channel request is mandatory to get the auth token, so that auth token will not be visible

Why JWT needs to be send in back channel request

Bec, JWT is just an Base64 encoded form of json user data, anyone can extract and easily decode the info without any key so this JWT auth token needs to be securely sent in back channel

After we hit this request, auth server will give us the login page so that user can enter his credentials, once credentials are verified, auth server provides the auth code (I don’t know why this instead auth server should have given us the auth token directly)





* Before we redirect the page to authorization server for authorization
  1. for every flow 1 separate hash code for that client secret will be assigned called pkce code verifier (43-123 characters long), sha-256 encode it and base 64 encode it and use this code while hitting auth server

ex:- http://autohrization-server.com/auth?response\_type=code & client\_Id=CLIENT\_ID&redirect\_uri=mm

* 1. when client is speaking with auth server, client must should the client id like how we show the id card issued by the company
  2. auth server will **generate the auth code** to it will send that auth code to client in front channel address bar

https://ticket-new.com/redirect?code=AUTH\_CODE\_HERE

* 1. now client wants the auth token and client will request auth server using auth code

<https://authorization-server.com/token/gratnt_type=authorization_code&code=AUTH_CODE_HERE&client_id=CLIENT_ID&client_secret=CLIENT_SECRET>

see client ask auth server with authcode as input

* 1. now auth server sends token in back channel In headers and client app uses this token while making api calls

{

“token\_type”:”Bearer”,

“access\_token”:’”dsfdfd”

“expires\_in”:3600,

“scope”:”photos”,

“refresh\_token”:”abcd”

}

* 1. Sometimes from auth server as response we will get refresh token, using that token also we can get auth token as response

https://authorization-server.com/token

grant\_type=**refresh\_token**&

refresh\_token=REFRESH TOKEN

client\_id=CLIENT ID&

client\_secret=CLIENT SECRET

//see here we are hitting auth server with refresh token as input to get auth token

Refresh token

Auth token will be used by ticket-new client apps

Auth code

Auth server

Access token types

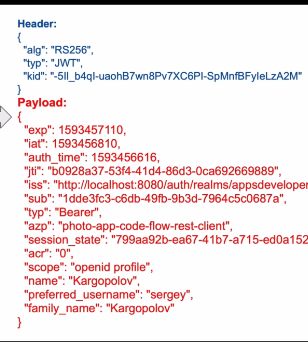
* 1. Identifier type access token- that issued token will be stored somewhere in db and will be referenced as below

|  |  |  |  |
| --- | --- | --- | --- |
| Access token | User\_id | Scope | Expires |
| Bbbgbgbsdf | Vvmani\_12 | Photos | 333333124ms |

And the token doesn’t contain any info

* 1. Self-contained type access token

It is a Json object which is base64 encoded it have – header,payload,signature



### Open ID token VS auth token

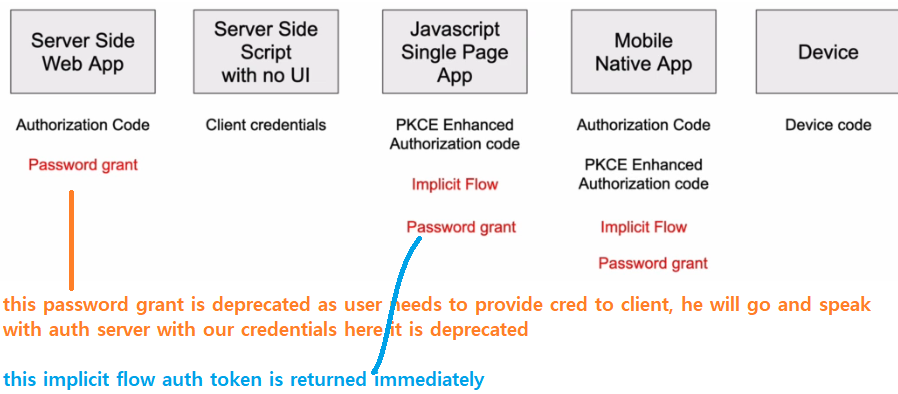
After validation - auth server gives only auth token, whereas open id connect gives- 2 tokens , auth token and id token which contains user information

## Grant types

It’s a way in which application gets an auth token from auth server,

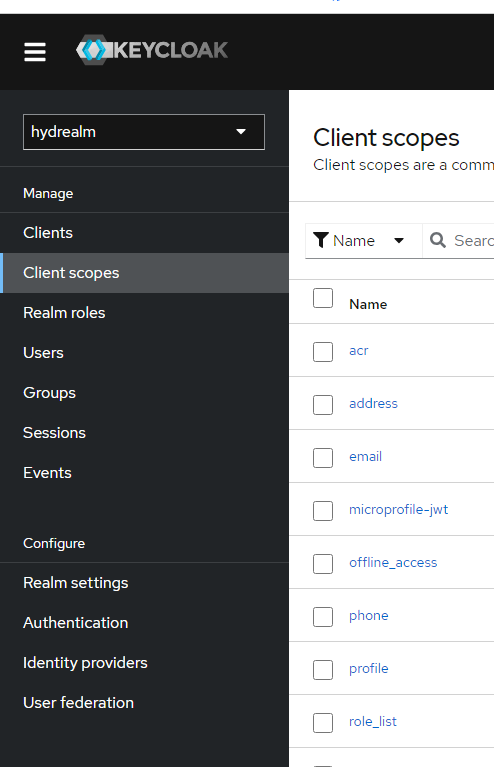
For a mobile device and java script app, server side app, for all these diff types of application the way they get the token from auth server differs

Means if appln is more secure like confidential client



Client scopes

All the client scopes are available In below path



**Refresh token grant type**

When a access token is expired, client can go and ask auth server to refresh it and again same token can be used

Like if our id card got expired they can extend right in same fashion

### **Authorization code grant**

Only confidential appln (If appln is capable of storing auth code /client secret securely in that server side app like java, not single page app) only those applications can use this grant

In auth code grant we use client id to get auth code

Auth code

Auth server

Client id

And provides client id, client secret (this can be passed only by confidential clients), auth code to auth server for auth token

Client id, client secret

Auth token

Auth server

Auth code

Remember, after entering our credentials auth server will verify and gives to us /redirects auth code in url address bar called front channel and hence fake appln can receive it

Whereas, during 2nd request auth token will be given back in response , not in address bar and hence it might not be easy to steal

Getting an auth code, auth token from auth server

Hit the url in browser and enter credentials and see the redirected url it contains the authcode

http://localhost:8080/realms/hydrealm/protocol/openid-connect/auth?client\_id=1dstr-wellsfargo&response\_type=code&state=edookati&scope=openid%20profile&redirect\_uri=http://localhost:8083/callback



Import this postman collection u can hit all these below requests

|  |  |
| --- | --- |
| 1. Initial authorization request  GET /authorization?  client\_id=aclientid&  scope=openid+email+profile&  response\_type=code&  redirect\_uri=http%3A%2F%2Fexample.org%2Fcallback | 2. Code exchange  POST /token  client\_id=aclientid& client\_secret=aclientsecret  ation\_code& grant\_type=authorization\_code&  code-issuedCodeFromGetCall&  redirect\_uri=http%3A%2F%2Fexample.org%2Fcallback |



Here response\_type must be “code” if u give auth\_code also it wont work

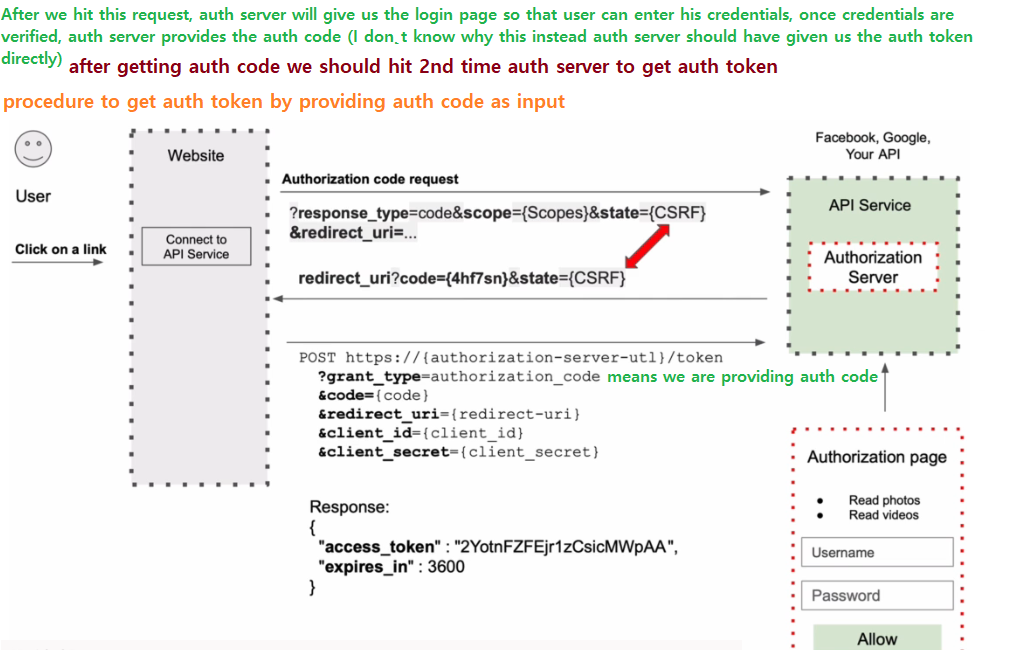
Ex:-

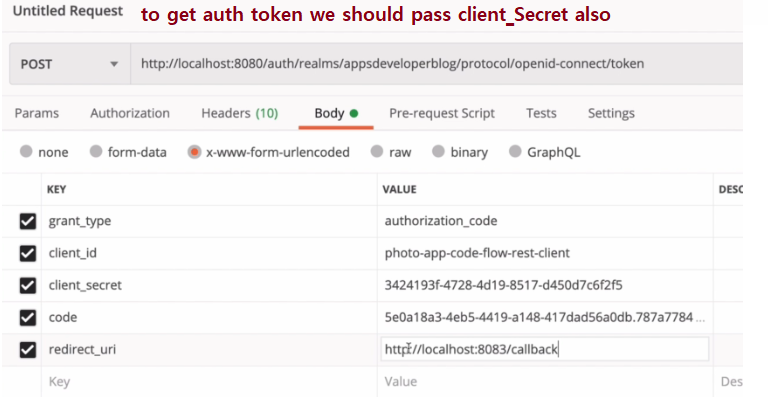
* 1. we should hit below url

http://localhost:8080/realms/hydrealm/protocol/openid-connect/auth?client\_id=1dstr-wellsfargo&response\_type=code&state=edookati&scope=openid%20profile&redirect\_uri=http://localhost:8083/callback

* 1. it will open a page & enter credentials after successful cred verification it will give below url which contains auth code

http://localhost:8083/callback?state=edookati&session\_state=f775d718-79fb-4b8f-9386-a5c211f9b566&iss=http%3A%2F%2Flocalhost%3A8080%2Frealms%2Fhydrealm&**code=92b3c4c9-20b1-4dbe-bc4c-11f220ed35c2.f775d718-79fb-4b8f-9386-a5c211f9b566.c978ad54-0fa0-4275-a242-23016c25d851**

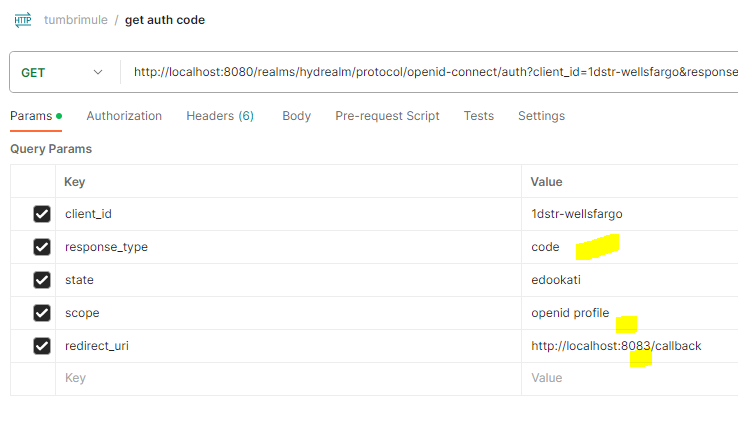




#### My practice screenshot

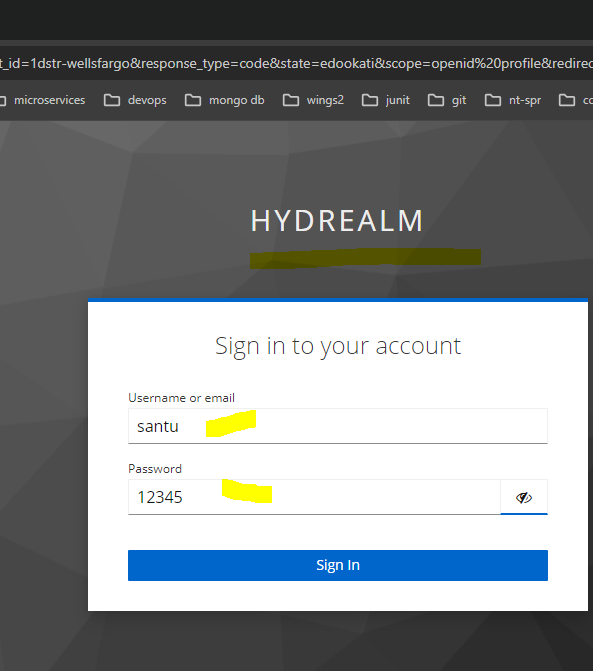
To get auth code, we should hit this url – here hydrealm is the one which I created

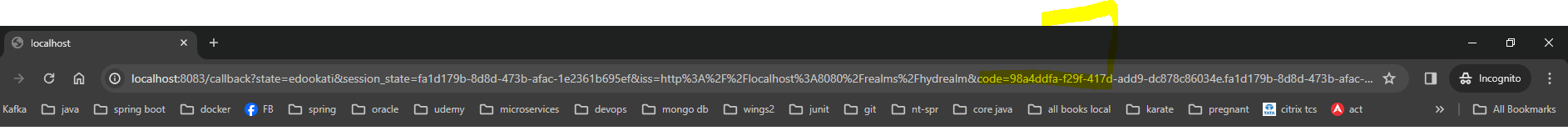
**http://localhost:8080/realms/hydrealm/protocol/openid-connect/auth?**



**http://localhost:8080/realms/hydrealm/protocol/openid-connect/auth?client\_id=1dstr-wellsfargo&response\_type=code&state=edookati&scope=openid profile&redirect\_uri=http://localhost:8083/callback**

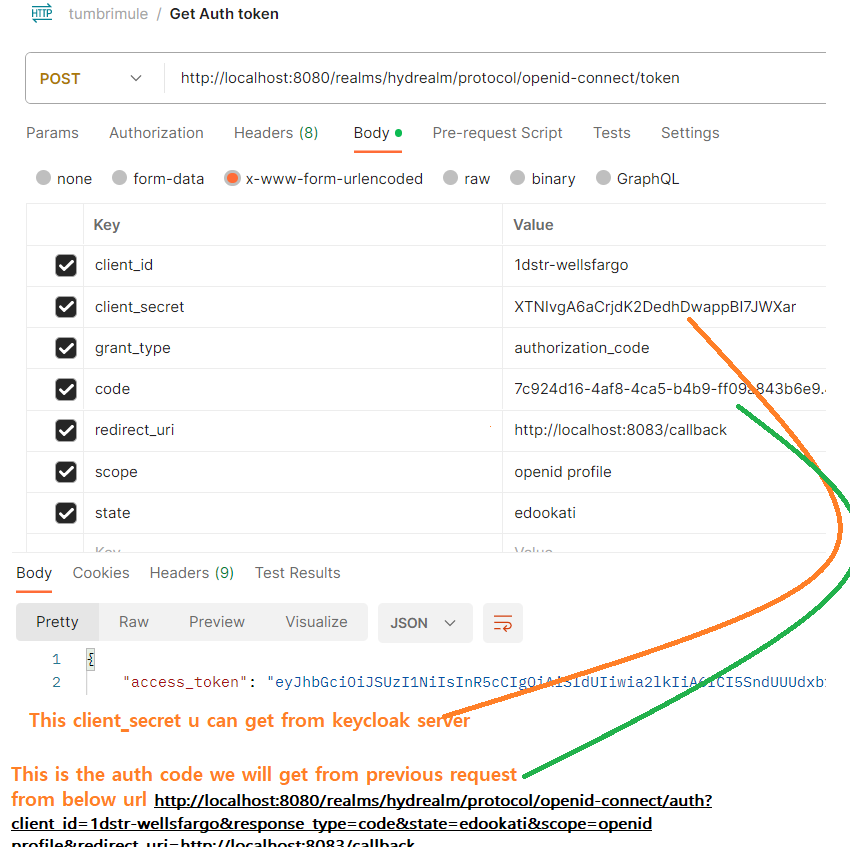
after keying all these details copy the url and paste in browser it will ask for client credentials ,once user enters the credentials & if auth is success then authcode will be generated as below in address bar





Once cred are verified then auth code will be redirected to us as above

To get auth token –we should pass both client id and client secret, below is redirect\_uri (its i not L or url) [**http://localhost:8080/realms/hydrealm/protocol/openid-connect/token**](http://localhost:8080/realms/hydrealm/protocol/openid-connect/token)

****

PKCE (proof key for code exchange) – enhance auth code

This is an extension for authorization code grant to protect against auth code injection & stealing auth codes

This PKCE is originally developed for public clients like single page apps/ mobile apps who can’t maintain confidential info like client secret, now this PCKE is standard for all

Public and confidential clients

Problem with authorization grant – fake legitimate appln can steal our auth code and with that they can get auth token

While getting auth code from auth server after cred validation, auth server redirects to our app having auth code, some fake appln having same url as ours,

Then auth code may go to that appln, other appln can steal our auth code, after that using that auth code that fake app can get the token (but to get that token that fake app should also have a client secret I think) and they can misuse that token

Now with PKCE, even if malicious app steals the auth code it cant get the token, because to get the token we should send authcode and original code verifier and that hacker doesn’t know the original code as we only knows it

PKCE helps prevent code interception attacks. OAuth 2.0 allows users to share their data securely between different applications, and PKCE provides an additional security layer on top of it

Here PKCE generates a key which will be understood by auth server

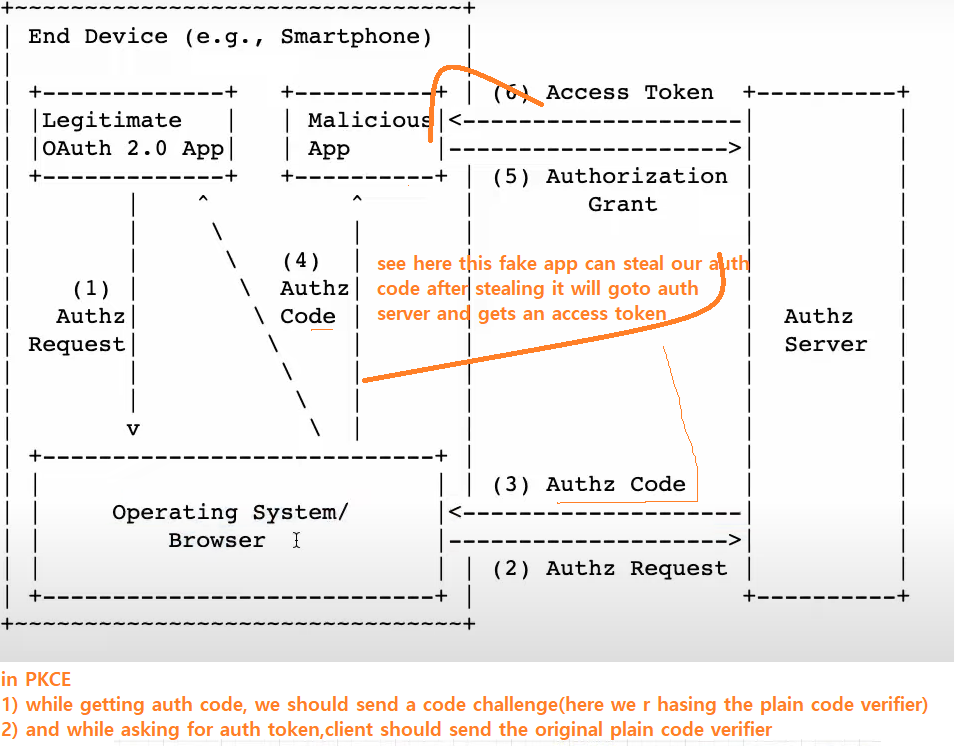
See

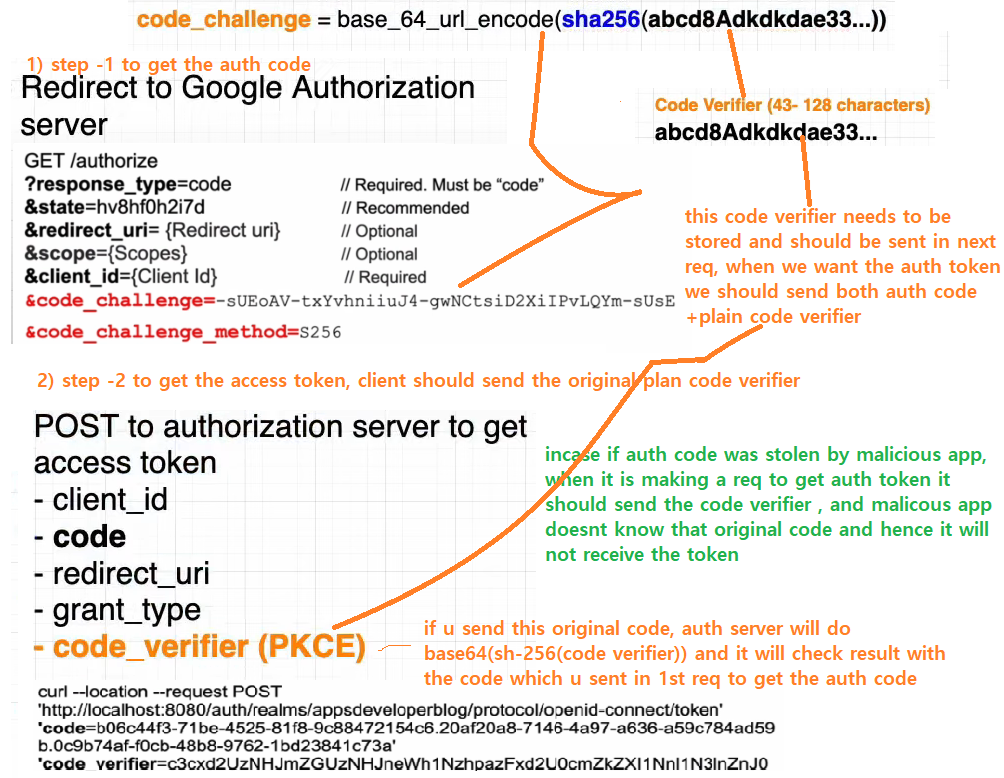
<https://www.authlete.com/developers/pkce/>

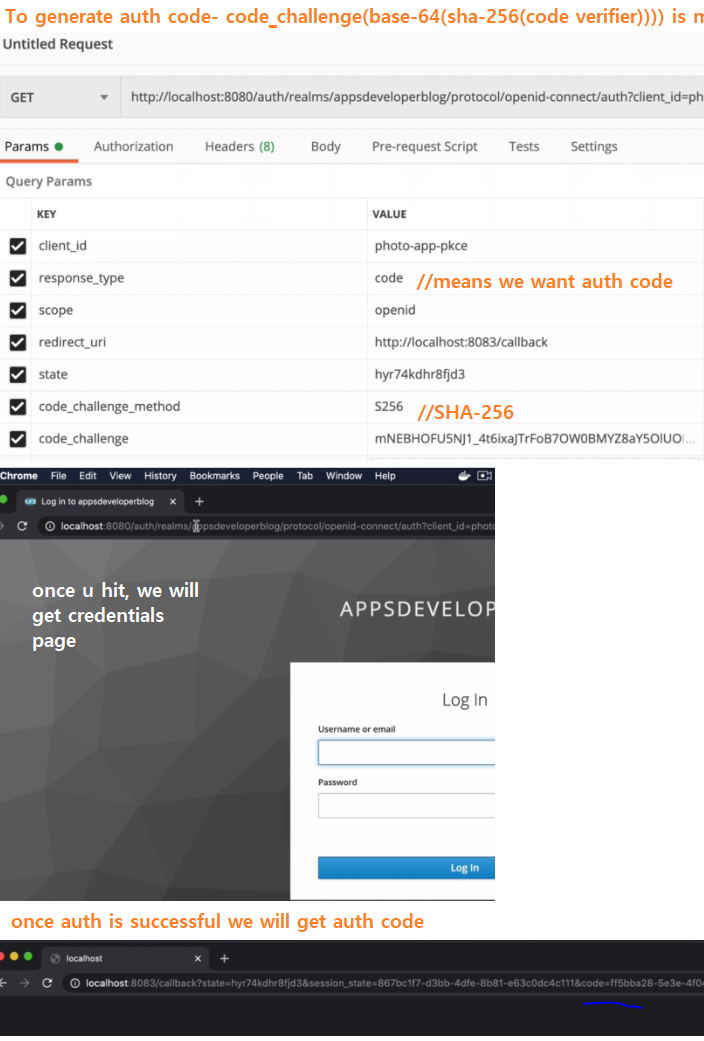
<https://pazel.dev/teach-me-pkce-proof-key-for-code-exchange-in-5-minutes>

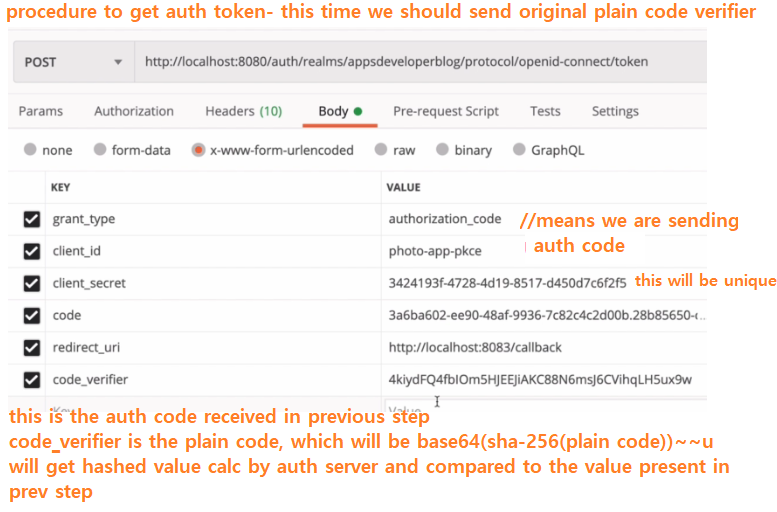
PCKE to generate auth code

Nore:- in url /auth/ will not be there for new keycloak versions

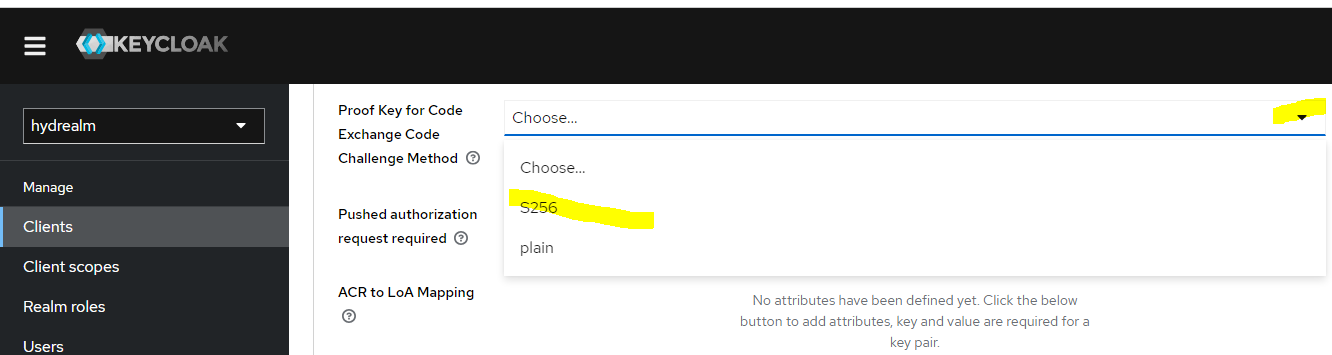








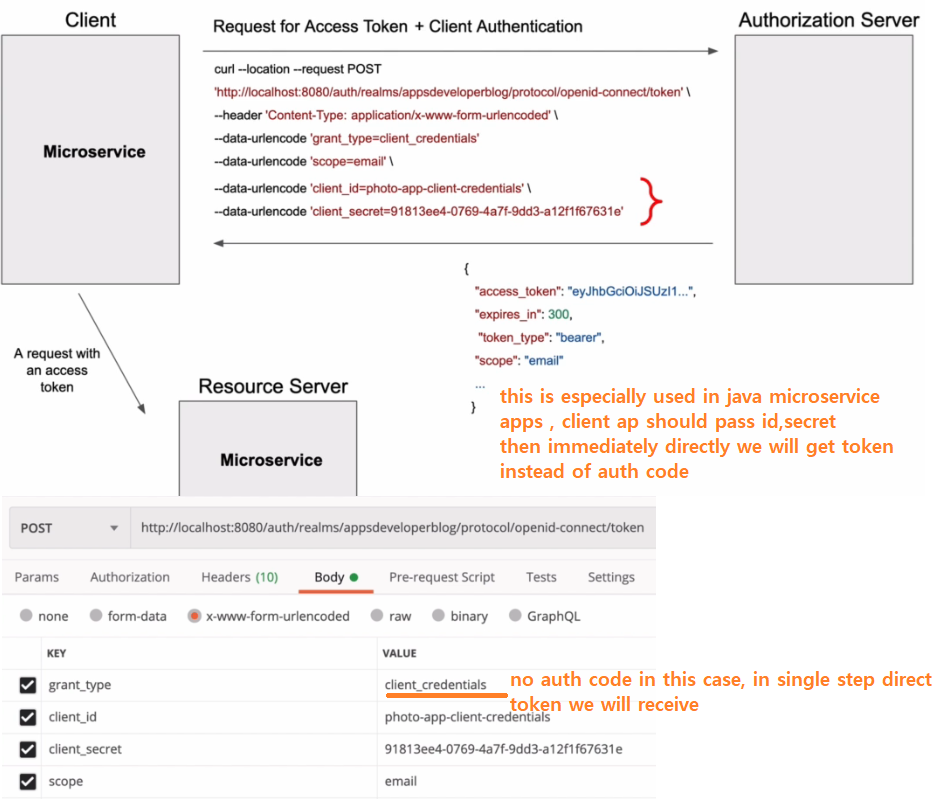
If u want to enable that PKCE in keycloak server then navigate to keycloak in below path as select s256



Client credentials (for java ms apps)

This is for machine to machine requests or java micro service to another java micro service, here there is no login page, no user nothing

Since there is not front channel as java app to java app is completely back channel request there is no need to auth code, we can directly get auth token



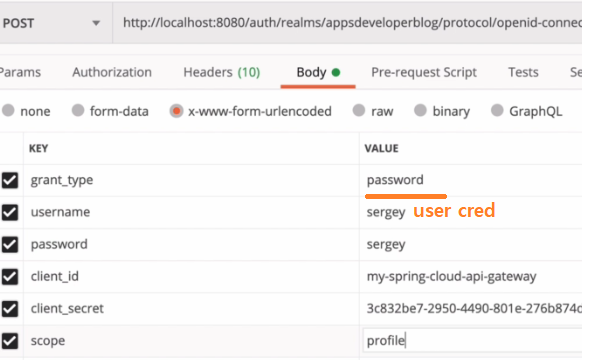
Password credentials

In this method, (resource owner) user should give his credentials to the client appln (ticket new), he will speak to auth server with our credentials, but this approach is almost old and nobody dares to give their

Credentials to other 3rd party appln

Ex:- if client ticket new app is talking to google auth server, user should give his google website credentials, now the client will take those credentials and talk to google auth server and gets the token and he even can misuse those too

When we will use this method:- if we believe client app then we can use this grant type



Refreshing access token



See, in below grant\_type is refresh token, to refresh the token also we need both client\_id, client\_Secret

curl --location --request POST 'http://localhost:8080/auth/realms/appsdeveloperblog/protocol/openid-connect/token' \

--header 'Content-Type: application/x-www-form-urlencoded' \

--data-urlencode 'grant\_type=refresh\_token' \

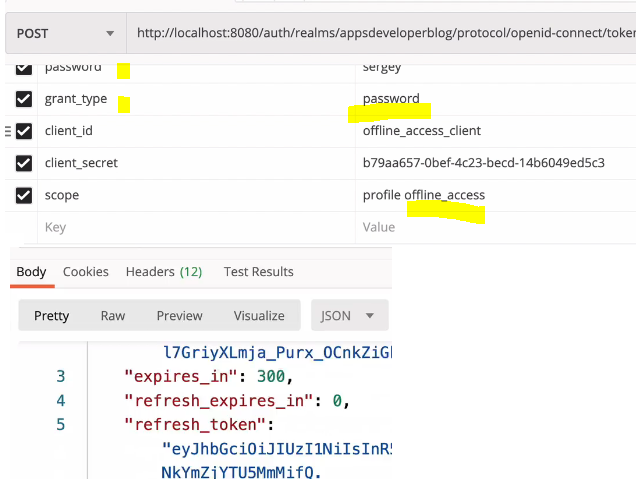
--data-urlencode 'client\_id=offline\_access\_client' \

--data-urlencode 'client\_secret=b79aa657-0bef-4c23-becd-14b6049ed5c3' \

--data-urlencode 'refresh\_token=eyJhbGciOiJIUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJlYWQyMDZmOS05MzczLTQ1OTAtOGQ4OC03YWNkYmZjYTU5MmMifQ.eyJleHAiOjE1OTUyNzMzMzIsImlhdCI6MTU5NTI3MTUzMiwianRpIjoiOWNhMDM3YWEtNDRmZi00MDYxLThmOTAtODBmMzAzMTJiMzNmIiwiaXNzIjoiaHR0cDovL2xvY2FsaG9zdDo4MDgwL2F1dGgvcmVhbG1zL2FwcHNkZXZlbG9wZXJibG9nIiwiYXVkIjoiaHR0cDovL2xvY2FsaG9zdDo4MDgwL2F1dGgvcmVhbG1zL2FwcHNkZXZlbG9wZXJibG9nIiwic3ViIjoiMWRkZTNmYzMtYzZkYi00OWZiLTliM2QtNzk2NGM1YzA2ODdhIiwidHlwIjoiUmVmcmVzaCIsImF6cCI6Im9mZmxpbmVfYWNjZXNzX2NsaWVudCIsInNlc3Npb25fc3RhdGUiOiI0YWNlNzlkNC0yOWU5LTQ1OGYtODI1My01N2UxYzE5NGY3NGIiLCJzY29wZSI6InByb2ZpbGUifQ.oeRL-UkkI3EnCDBsbQDrcRD5lE8\_LidDiKwEIuJRFJM'

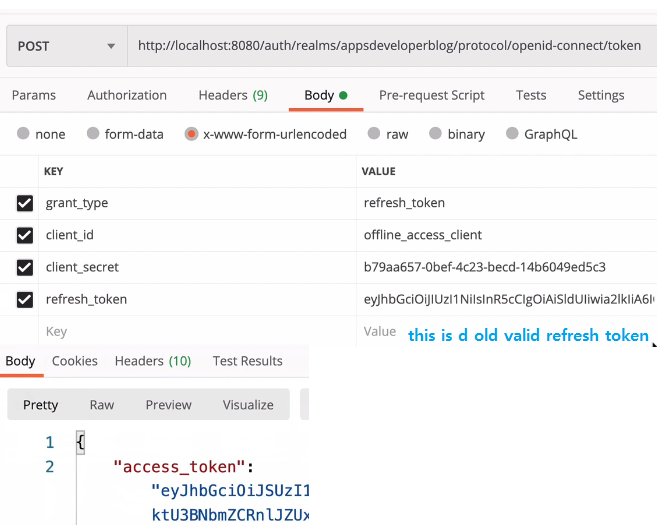
Requesting a refresh token that never expires

When the scope is offline\_access then u will get a refresh token that will never expires and u can see value as o for “refresh \_Expires\_in”



Refreshing a access token with help of refresh token

If we have a refresh token which is live, with that we can ask auth server to refresh the auth token



Creating Resource server with token validation

We will be developing a REST API that is also called Resource server

REST API will have following responsibilities

Auth server like okta or keycloak auth server

Postman or client app

1. It should send the token in header

Resource server API tasks

1. Received auth token from header & validated with help of auth server
2. If validated then send the data to postman
   1. While hitting that REST API or resource server, postman/ client app should pass the auth token (which we got from auth server) in header,
   2. When resource server receives that request then That resource server should communicate with jwt auth server and validate that token that is passed in header

Once success then only resource server should send the response to client

*#spring.security.oauth2.resourceserver.jwt.issuerUri=http://localhost:8080/realms/hydrealm*spring.security.oauth2.resourceserver.jwt.jwkSetUri=http://localhost:8080/realms/hydrealm/protocol/openid-connect/certs

Any one property is fine among above 2

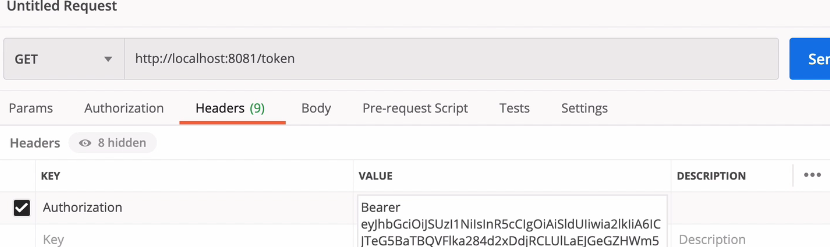
We only have to tell to resource server (our API) about where is auth server, so that it will take & provide our token to auth server for validating that

<dependency>

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

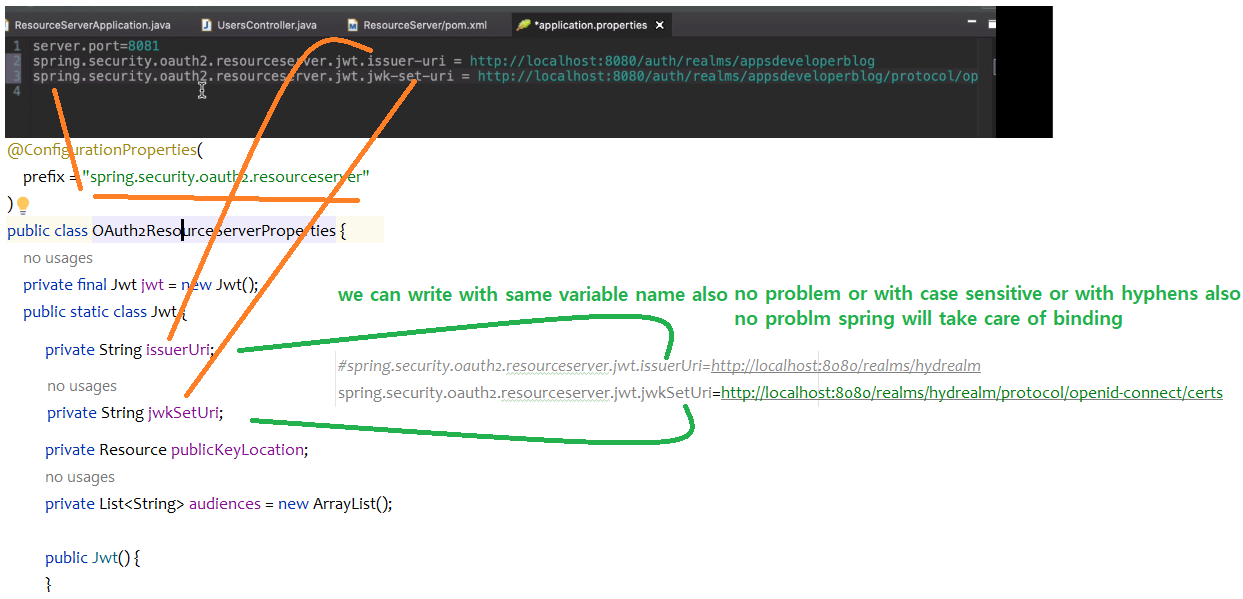
<artifactId>spring-boot-starter-oauth2-resource-server</artifactId>

</dependency>



What is happening internally?

* 1. The token which we sent in header, will be taken by spring framework automatically and spring will verify this token with auth server,
  2. our responsibility is just to pass the token in header and
  3. we should tell the url of auth server then only this resource server will go & talk to auth server and validate token
  4. to see all the authentication information



Scope / permission based access

Giving token means- with that token we should not access all apps like drive, email,photos, calendar .. token should have limited powers/ limited scope

Scope is a mechanism in OAuth 2.0 to limit an application's access to a user's account. An application can request one or more scopes (like photos, email), Hence while accessing

This information is then presented to the user in the consent screen, and the access token issued to the application will be limited to the scopes granted.

Hence **scope must be mentioned** while showing auth page or while getting auth token in postman

Ex:- scope “open id” means along with auth token u will get id token (which have some identity information about the user)

Opened connect standard scope



**in the issued token u can see the scope of that token**

|  |  |
| --- | --- |
| Scope:- profile | If scope is profile then client will be able to access all these below details  name, family\_name, given\_name, middle\_name, nickname, preferred\_username, profile, picture, website, gender, birthdate, locale, and zoneinfo, updated\_at |
| Email | Email, email\_verified  Under this scope client will be able to access email id and he can check if email id is verified or not |
| Address | formatted,street\_address, locality, region,postal\_code,country |
| Phone | Phone, phone\_number\_verfied |

How to achieve scope based access

* 1. configure the controller endpoints with certain scope ex:- only tokens which was issued for the scope “profile” can access this endpoint
  2. while getting the auth code itself mention the scope and get the access so that internally that scope is mapped to that token

Security class configurations

Note: - initially we used to extend our class with websecurityConfigurerAdapter whereas in latest spring versions, no need to extend that class instead **we can create bean definitions** for SecurityFilterChain will be enough

#### Provide configurations in config class

@Configuration  
@EnableWebSecurity  
public class SecurityConfig {  
  
 @Bean  
 public SecurityFilterChain configure(HttpSecurity http) throws Exception {  
 JwtAuthenticationConverter jwtAuthenticationConverter = new JwtAuthenticationConverter();  
*// jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(new KeycloakRoleConverter());* http  
 .authorizeHttpRequests(auth -> {  
 auth.requestMatchers(HttpMethod.*GET*, "/user").hasAuthority("SCOPE\_profile")  
 // Here if u see url "/user" can only be accessed if that token have scope "profile", means while issuing the authcode & token user should mention the profile scope .anyRequest().authenticated();  
 }  
 )  
 .oauth2ResourceServer(  
 oauth2 -> oauth2.jwt(jwt -> jwt.jwtAuthenticationConverter(jwtAuthenticationConverter)));  
 return http.build();  
 }  
}

@GetMapping("/user")  
public ResponseEntity<String> user(){  
 return ResponseEntity.*ok*("Heloo i am a secured user");  
}

Older versions:- in old way we used to tightly couple by extending with spring related class and it is not a bean , now we will just create bean definitions

@EnableWebSecurity

public class SpringSecurityDemo extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http

// Enable basic authentication

.httpBasic()

.and()

// Secure all requests by default

.authorizeRequests()

.antMatchers("/login").permitAll() // Allow access to login endpoint

.anyRequest().authenticated()

.and()

// Configure a simple login form (replace with your actual login logic)

.formLogin()

.loginPage("/login")

.usernameParameter("username")

.passwordParameter("password")

.defaultSuccessUrl("/")

.permitAll();

}

@Override

protected void configure(AuthenticationManagerBuilder auth) throws Exception {

// In-memory authentication (replace with your user store)

auth.inMemoryAuthentication()

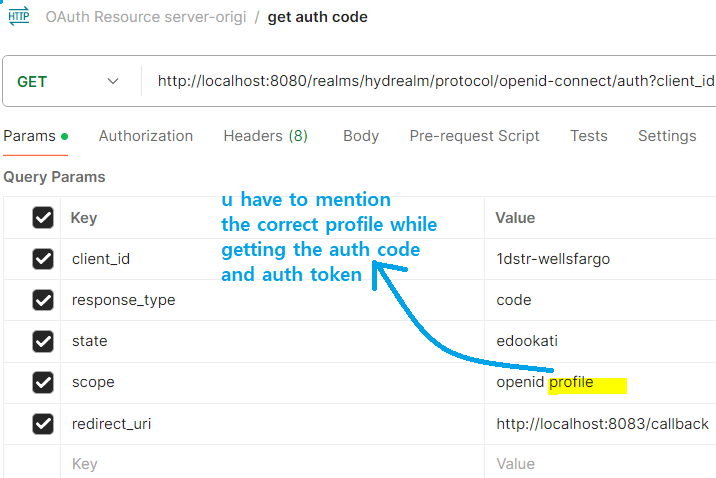
.withUser("user").password("{noop}password").roles("USER")

.and()

.withUser("admin").password("{noop}admin").roles("ADMIN");

}

Mention the scope while asking for auth code / auth token



Role based access



**In the issued token u can see all the roles**, in auth server (keycloak or okta) we can add a new role for each and every user

In this privilege is nothing but authority ex:- I have the privilege to delete the access something like that

A Role is nothing but collection of authorities ROLE = Authority

|  |  |  |
| --- | --- | --- |
| Role: User and his privileges or authorities | Role:- Admin & his privileges or authorities | Role:- Super admin |
| View his profile | View his & others profile | View his & others profile |
| View others profile | Edit his & others and delete other profile | Edit his & others and delete other profile |
| Edit his own profile |  | Edit or delete other admin profile |

In wells fargo I am playing developer and engagement owner role and auditor and other roles

|  |  |
| --- | --- |
| Spring method | Analysis |
| hasRole(“ADMIN or any other role”)  hasRole("developer") | Since the method name already have role this method will add “ROLE\_” automatically to provided role |
| hasAuthority(“ROLE\_ADMIN or any other role”) | Since this method name doesn’t mention whether this is a role or scope we only have to tell if it is a role or scope |
| hasAuthority("SCOPE\_profile") | See here also as method name doesn’t depict anything we have told that it is a scope |

Note:- We should create a converter class to extract the roles from the token

@Bean  
 public SecurityFilterChain configure(HttpSecurity http) throws Exception {  
 JwtAuthenticationConverter jwtAuthenticationConverter = new JwtAuthenticationConverter();  
 jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(new KeycloakRoleConverter());  
 http  
 .authorizeHttpRequests(auth -> {  
*// auth.requestMatchers(HttpMethod.GET, "/user").hasAuthority("SCOPE\_profile")* auth.requestMatchers(HttpMethod.*GET*, "/user").hasRole("developer")  
 *// Here if u see url "/user" can only be access if that token have scope "profile", means while issuing the token user should mention the profile scope  
 //Here if u see url "/user " can only be access by developer* .anyRequest().authenticated();  
 }  
 )  
 .oauth2ResourceServer(oauth2 -> oauth2.jwt(jwt -> jwt.jwtAuthenticationConverter(jwtAuthenticationConverter)));  
 return http.build();  
 }  
}

#### Converter class to fetch the roles from token

import org.springframework.core.convert.converter.Converter;  
import org.springframework.security.core.GrantedAuthority;  
import org.springframework.security.core.authority.SimpleGrantedAuthority;  
import org.springframework.security.oauth2.jwt.Jwt;  
  
public class KeycloakRoleConverter implements Converter<Jwt, Collection<GrantedAuthority>> {  
 @Override  
 public Collection<GrantedAuthority> convert(Jwt jwt) {  
 Map<String, Object> realmAccess = (Map<String, Object>) jwt.getClaims().get("realm\_access");  
  
 if (realmAccess == null || realmAccess.isEmpty()) {  
 return new ArrayList<>();  
 }  
  
 Collection<GrantedAuthority> returnValue = ((List<String>) realmAccess.get("roles"))  
 .stream().map(roleName -> "ROLE\_" + roleName)  
 .map(SimpleGrantedAuthority::new)  
 .collect(Collectors.*toList*());  
  
 return returnValue;  
 }  
}

The below token which is issued by auth server if u decode it using jwt.io we can clearly see the roles of the user or subject

eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI5SndUUUdxb1E5QlVwLWdiNUt2LU9zTnQ4cGlON0cxc0FYRllqX2stZklzIn0..mqLkjX2GKVeP\_bmo0QG3KYCdJc8czi5ZCUsuudQufE3LXN22jFKwHFjA6BVWkKKXojn9TUTaLvVpiG6wSm2Ftwvj8qLdos759yR33Z4RtyEUl8zlM7kOocdw3NB7e35OL4H1KiwtLJX1\_ukKbacjPA6JLeL3\_XFHLFcm2YLoShvMzd6SwyfBjkl3pvcluV93UJ\_frpqrja3318klYjVi1HLEsLPLgvZqFuJuit\_pI4AAXJTylArMnkAgABhPucicUCIg0QQoGsMHEJFaHUzNSIDqMS4WLBAnC1MsKXHzU9KZTz3aV5\_tnYYLatRPPqrvE7T\_CYnv1dRfVDg2Mgq9mg

"realm\_access": {

"roles": [

"default-roles-hydrealm",

"offline\_access",

"developer",

"uma\_authorization"

]

},

Spring expression based access control

|  |  |
| --- | --- |
| hasRole([role]) | Returns true if the current principal has the specified role. |
| hasAnyRole([rolel, role2]) | Returns true if the current principal has any of the supplied roles (given as a comma-separated list of strings) |
| principal | Allows direct access to the principal object representing the current user |
| authentication | Allows direct access to the current Authentication object obtained from the securityContext |
| permitAll | Always evaluates to true |
| denyAll | Always evaluates to false |
| isAnonymous() | Returns true if the current principal is an anonymous user |
| isRememberMe() | Returns true if the current principal is a remember-me user |
| isAuthenticated() | Returns true if the user is not anonymous |
| isFullyAuthenticated() | Returns true if the user is not an anonymous or a remember-me user |

Note:- when u are using authority u have to clearly tell whether it is a role or whether it is a scope

|  |  |
| --- | --- |
| @PreAuthorize(“hasRole(‘developer’)”) | @PreAuthorize(“hasAuthority(‘ROLE\_developer’)”)  // If u use has Authority then u have to specify whether it is a role or scope  @Secured(“ROLE\_developer”)  //can we mention scope also in @Secured annotation? |
|  |  |
| .hasAuthority("SCOPE\_profile")  .hasAnyAuthority("ROLE\_developer") | Since method name is authority we have to tell whether its role or scope |
| .hasRole("developer")  .hasAnyRole("developer","user") | Since the method name already has role no need to mention the keyword ROLE again |

#### Method level security

We can implement this method level security with 2 anno

@PreAuthorize in this first authorization will takes places, if he is not authorized then method wont get executed at all, whereas in @PostAuthorize() once entire method code is executed then at last condition will be verified , if condition fails then output alone will not be returned

|  |  |
| --- | --- |
| Mandatory annotation to enable other annotations  @EnableMethodSecurity(  securedEnabled = true,prePostEnabled = true  ) | * + 1. To enable the @Secured & @PreAuthorize @PostAuthorize annotations we have to enable/add this   If u don’t enable / write this anno then above 2 anno won’t work   * + 1. We have to write converter class to fetch the realms from the token |
| @Secured("ROLE\_admin")  @Secured({ "ROLE\_USER", "ROLE\_ADMIN" }) | This secured annotation supports only for role not for scope |
| @PreAuthorize("hasAuthority('ROLE\_developer')")  @PreAuthorize("hasRole('ADMIN') or hasRole('MANAGER')") | This annotation supports both role and scope |
| @PreAuthorize("hasAuthority('ROLE\_admin') or #id == #mytoken.subject ")  @PreAuthorize("hasRole('admin') or #id == #mytoken.subject ") | @GetMapping("/admin/{id}") @PreAuthorize("hasAuthority('ROLE\_admin') or #id == #mytoken.subject ")  //@PreAuthorize("hasRole('admin') or #id == #mytoken.subject ")  *//means this endpoint is accessible only by admins or if we pass and id which is equal to the decoded tokens "sub" value then also we can hit this url , here mytoken & id are my local variables*  public ResponseEntity<String> admin(@PathVariable String id, @AuthenticationPrincipal Jwt mytoken){  return ResponseEntity.*ok*("Heloo i am an admin with id--> "+ id); }  url  <http://localhost:8083/admin/24846346-db03-439f-9f5b-0864e89010ef>  in jwt token decoded payload matches with below info  { "sub": "24846346-db03-439f-9f5b-0864e89010ef" } |
| @PreAuthorize("hasAuthority('SCOPE\_profile')") | This scope didn’t work, but token have the profile scope & I decoded the token and saw the profile scope in jwt.io but still it didn’t work |
| @PostAuthorize("returnObject.getStatus() == 'APPROVED' or hasRole('APPROVER')") | @PostAuthorize("returnObject.getStatus() == 'APPROVED' or hasRole('APPROVER')")  public Order approveOrder(Long orderId) {  // Logic to fetch and potentially modify the order  Order order = orderRepository.findById(orderId);  order.setStatus("APPROVED");  orderRepository.save(order);  return order;  } |
| @PostAuthorize("hasRole('admin') or #id == #mytoken.id")  //since it is a post auth 1st method will be executed later if condition match return type will be passed else return type will not be passed at all | @GetMapping("/postAuth/{id}") @PostAuthorize("hasRole('admin') or #id == #mytoken.id") *//means this endpoint will give us return value only if this is accessed by admins or if u pass an id == {"jti": "e7048025-4b72-4257-9f39-e86ccc4142ae"} in decoded jwt token incase* public ResponseEntity<String> postAuth(@PathVariable String id, @AuthenticationPrincipal Jwt mytoken){  System.*out*.println("HI i am in post authorize code is already executed with jwt id " +mytoken.getId());  return ResponseEntity.*ok*("Heloo i am from post auth with id--> "+ id); } |

