Nm, .m What is HTTP and HTTPS ? why do we need HTTPS ? explains about HTTP and HTTS and SSL and TLS versions ?

What does a HTTP request and response contains ,

How does HTTPS works ? what is SSL and TLS ? what is public key and private key ? what is symmetric and asymmetric encryption key ?

What does a HTTPS certificate contains? how to generate a HTTPS certificate? What is trust store ? and key store ?

How does TLS work ?

self signed certificate vs ca signed certificate ?

Algorithms and encryptions used in HTTPS ?

watch this video <https://youtu.be/LJDsdSh1CYM>

**TLS/SSL Overview:**

TLS (Transport Layer Security) and SSL (Secure Sockets Layer) are protocols used to secure communication over the internet, ensuring data is encrypted, authentic, and safe from tampering.

**How It Works:**

1. **Client Initiates Connection:** Your browser (client) connects to a server (e.g., a website) using HTTPS.
2. **Server Responds with Certificate:** The server sends its SSL/TLS certificate to prove its identity. This certificate contains the server’s public key.
3. **Client Verifies Certificate:** Your browser checks if the certificate is valid and issued by a trusted Certificate Authority (CA). (verify certificate chain, signature, check expiry date, domain name ,root-issuer)
4. **Key Exchange:**
   * The client generates a random value, encrypts it with the server's public key, and sends it to the server.
   * The server decrypts it with its private key. This value is used to create a session key.
5. **Session Key Established:** Both the client and server use this session key to encrypt and decrypt the data they send to each other.
6. **Secure Communication:** All data exchanged between your browser and the server is now encrypted, ensuring privacy and security.

**What is Mutual TLS (mTLS)?**

Mutual TLS (mTLS) is an extension of TLS where both the client and the server authenticate each other using certificates, providing an additional layer of security. It ensures that both parties in a communication are trusted.

**How mTLS Works in Spring Boot Microservices:**

1. **Certificate Setup:**
   * **Server Certificate:** Each microservice server has its own certificate and private key, typically issued by a trusted Certificate Authority (CA).
   * **Client Certificate:** Each client (which could be another microservice) also has its own certificate and private key.
2. **Two-Way Authentication:**
   * **Client Authentication:** When the client (microservice) makes a request to the server (another microservice), it presents its client certificate.
   * **Server Authentication:** The server checks this certificate against its list of trusted CAs to verify the client’s identity.
   * **Server Certificate:** The server also presents its certificate to the client.
   * **Client Verification:** The client verifies the server’s certificate against its trusted CA list.
3. **Secure Communication:**
   * Once both parties are authenticated, they establish a secure, encrypted connection using a session key derived during the handshake process.
   * All data exchanged between the microservices is now encrypted and both parties are verified.

**Setting Up mTLS in Spring Boot:**

1. **Generate Certificates:**
   * Use tools like OpenSSL to generate the required certificates for both the client and server.
2. **Configure Spring Boot:**
   * **Server Configuration:** In the server’s application.properties or application.yml, configure the server to require client authentication.
   * **Client Configuration:** Configure the client to present its certificate when making requests.

Example configuration for the server (application.yml):

yaml

Copy code

server:

ssl:

key-store: path/to/server-keystore.jks

key-store-password: password

key-alias: server-alias

trust-store: path/to/truststore.jks

trust-store-password: password

client-auth: need

Example configuration for the client (application.yml):

yaml

Copy code

ssl:

key-store: path/to/client-keystore.jks

key-store-password: password

key-alias: client-alias

trust-store: path/to/truststore.jks

trust-store-password: password

1. **Test and Deploy:**
   * Deploy the microservices with mTLS configured. Test to ensure that both microservices can successfully authenticate each other and communicate securely.

**Benefits of mTLS in Microservices:**

* **Enhanced Security:** Both parties are authenticated, preventing unauthorized access.
* **Data Integrity and Privacy:** Data is encrypted, ensuring it cannot be intercepted or tampered with.

In summary, mTLS in Spring Boot microservices involves both the client and server authenticating each other using certificates, followed by secure, encrypted communication. This setup is critical in environments where strong security is required, such as financial systems or internal enterprise applications.

**TLS/SSL VS Mutual TLS**

Mutual TLS (mTLS) and TLS/SSL are related but serve different purposes. Here’s a comparison to clarify their differences:

**TLS/SSL (Transport Layer Security / Secure Sockets Layer):**

* **Purpose:** TLS (and its predecessor SSL) are cryptographic protocols designed to provide secure communication over a network. They ensure data confidentiality, integrity, and authenticity between a client and server.
* **Operation:** When a client (like a web browser) connects to a server (like a website), the server presents its SSL/TLS certificate to the client. The client verifies the certificate and then establishes an encrypted connection. This ensures that the data exchanged between the client and server is secure.
* **One-Way Authentication:** In standard TLS/SSL, only the server is authenticated to the client. The client does not present a certificate to the server.

**Mutual TLS (mTLS):**

* **Purpose:** mTLS is an extension of TLS where both the client and server authenticate each other. This provides an additional layer of security because both parties in the communication are verified.
* **Operation:** In addition to the server presenting its certificate to the client (as in standard TLS/SSL), the client also presents its own certificate to the server. The server verifies the client’s certificate, ensuring that the client is also authenticated.
* **Two-Way Authentication:** Both the client and the server have certificates and verify each other’s certificates. This is particularly useful in scenarios where enhanced security is required, such as in enterprise environments or for securing internal APIs.

**Summary:**

* **TLS/SSL:** Provides secure communication and one-way authentication (server only).
* **mTLS:** Extends TLS/SSL to include two-way authentication (both server and client).

**Example Use Cases:**

* **TLS/SSL:** Securing websites (HTTPS), encrypting emails (STARTTLS), securing API communications where server authentication is sufficient.
* **mTLS:** Securing internal microservices communication, ensuring that both parties in a communication are trusted in enterprise environments, and in financial transactions requiring strong mutual authentication.

In summary, while both TLS and mTLS are used for secure communication, mTLS adds an additional layer of security by requiring mutual authentication.

**KeyStore VS TrustStore**

**Keystore:**

* **Purpose:**
  + Stores the private keys and the corresponding certificates that the application uses to identify itself to others (e.g., a server presenting its certificate to a client).
  + It can also store the public keys of others, but its primary role is to manage the credentials of the application itself.
* **Contents:**
  + Private keys, along with their associated public key certificates.
  + Sometimes, it may also contain CA certificates.
* **Use Case:**
  + When a server needs to authenticate itself to clients (e.g., HTTPS server).

**Truststore:**

* **Purpose:**
  + Stores the public certificates of external parties (e.g., Certificate Authorities (CAs) and trusted servers) that the application trusts.
  + Used to validate the certificates presented by others (e.g., a client verifying a server's certificate).
* **Contents:**
  + Trusted CA certificates and public certificates of other parties.
* **Use Case:**
  + When an application (client or server) needs to verify the authenticity of certificates presented by others.

**Summary:**

* **Keystore:** Contains your application's private keys and certificates, used to prove its identity.
* **Truststore:** Contains certificates from external entities that your application trusts.

**Example in a Spring Boot Application:**

* **Keystore:** A server might use a keystore to store its SSL certificate and private key, which it uses to establish HTTPS connections.
* **Truststore:** The same server might use a truststore to store CA certificates, allowing it to verify the certificates of clients or other servers it communicates with.

**What is JWT (JSON Web Token)?**

JWT (JSON Web Token) is a compact, URL-safe token format used for securely transmitting information between parties as a JSON object. It’s commonly used for authentication and authorization in web applications, including Spring Boot microservices.

**Structure of JWT:**

A JWT consists of three parts, separated by dots:

1. **Header:** Contains the token type (JWT) and the signing algorithm (e.g., HMAC SHA256).
2. **Payload:** Contains the claims, which are statements about the user (like user ID, roles, and expiration time).
3. **Signature:** Ensures the token hasn’t been altered. It's created by encoding the header and payload, then signing it using a secret key or a private key.

Example: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvbiBEb2UiLCJpYXQiOjE1MTYyMzkwMjJ9.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c

**Flow of JWT in Spring Boot Microservices:**

1. **User Authentication:**
   * The client (e.g., a web or mobile app) sends a login request with credentials to an authentication microservice.
   * The microservice verifies the credentials against the database.
2. **JWT Generation:**
   * If the credentials are valid, the authentication microservice generates a JWT.
   * The JWT contains claims (e.g., user ID, roles) and is signed with a secret key to prevent tampering.
   * The token is then sent back to the client.
3. **Client Stores JWT:**
   * The client stores the JWT (typically in local storage or a cookie) and includes it in the header of subsequent requests to other microservices.
4. **JWT Validation:**
   * When the client makes a request to a secured microservice, it includes the JWT in the Authorization header (e.g., Authorization: Bearer <token>).
   * The microservice extracts the JWT from the request and validates it by:
     + Verifying the signature using the secret key or public key.
     + Checking the token’s expiration time and claims.
   * If the token is valid, the request is processed; otherwise, it's rejected.
5. **Authorization:**
   * The microservice checks the claims in the JWT (like user roles) to authorize the user’s access to specific resources or operations.
6. **Response to Client:**
   * If everything is valid, the microservice processes the request and sends the response back to the client.

**Benefits of Using JWT:**

* **Stateless:** JWTs are self-contained, meaning all the information needed for authentication is within the token, eliminating the need to store session data on the server.
* **Scalability:** Since tokens are stateless, they are ideal for microservices architectures where each service can independently validate the token.
* **Security:** The token is signed and can include expiration times to prevent misuse.

**Example in Spring Boot:**

In a Spring Boot microservices setup, you would typically configure JWT authentication in a centralized authentication service and then propagate the JWT to other microservices for validation. Spring Security can be used to handle JWT validation and authorization.

**Flow Summary:**

1. Client logs in and receives a JWT.
2. Client includes JWT in headers of requests to microservices.
3. Microservices validate JWT and authorize the request.
4. If valid, the service processes the request and responds.

This flow ensures secure, stateless authentication across multiple services.

How will you communicate are authenticate your self by other microservice, we send a client and secrete key to identity service that will send us a JWT token which we use to send to other microservices ( how we get secrete will raise a request during on boarding process then we get this key)

**Use Case Breakdown:**

1. **Microservice Interaction:**
   * You have a microservice (let's call it Service A) that needs to interact with other microservices (Service B, Service C, etc.).
   * Each time Service A needs to call another service, it must authenticate itself.
2. **JWT for Authentication:**
   * **JWT (JSON Web Token)** is a secure way to represent claims between two parties.
   * To interact with the other microservices, Service A needs to present a JWT token as proof of its identity and permissions.
3. **Obtaining the JWT:**
   * **Identity Service**: Service A calls an Identity Service (let's call it Auth Service) to obtain a JWT.
   * **Credentials**: Service A provides its client\_id and client\_secret to the Auth Service.
   * **Token Issuance**: The Auth Service authenticates Service A using these credentials and issues a JWT.
   * **Token Usage**: Service A then uses this JWT to authenticate its requests to Service B,

**OAuth2.0:** OAuth 2.0 stands for \*Open Authorization Protocol. It is primarily **used to secure application resources** **by allowing limited access to resources by using access tokens**. These tokens are issued by an authorization server and represent the user's authorization to access specific resources**.**

**OAuth2.0 Roles or Key Components:**

* **Resource Owner:** The user or system that owns the protected resources and can grant access to them. (my Self)
* **Client:** The client is the system that requires access to the protected resources. To access resources, the Client must hold the appropriate Access Token. (HackerRank)
* **Authorization Server:** This server receives requests from the Client for Access Tokens and issues them upon successful authentication and consent by the Resource Owner. The authorization server exposes two endpoints: the Authorization endpoint, which handles the interactive authentication and consent of the user, and the Token endpoint, which is involved in a machine-to-machine interaction. (GitHub Auth Server)
* **Resource Server:** A server that protects the user’s resources and receives access requests from the Client. It accepts and validates an Access Token from the Client and returns the appropriate resources to it. (GitHub User Info Service)
* **Flow explanation:**
* 1. User and Client Interaction: The flow starts when the user interacts with the client (for example, a web application). The client then redirects the user to the \*\*authorization server\* to obtain authorization.
* 2. \*Authorization Code Flow: the user is redirected to the authorization server's login page, where they log in and grant permission. After that, the authorization server provides an \*authorization code \* to the client (redirected to a specified redirect URI). (**Authorization server provides Client with either an Authorization Code or Access Token, depending on the grant type,) if Authorization server provides Access token next step(**Exchanging the Authorization Code**) is not required (Authorization code is recommended as its more secure)**
* 3. \*Exchanging the Authorization Code: The client then exchanges this \*\*authorization code\* with the authorization server for an \*access token**\*. This exchange includes the authorization code, client ID, client secret**, and the redirect URI.
* 4. \*Access Token Usage: Once the client obtains the access token, it uses this token to access resources from the \*\*resource server\* on behalf of the user.
* 5. \*Data Retrieval\*: The resource server verifies the access token and, if valid, provides the requested resources/data to the client, which then returns the data to the user.

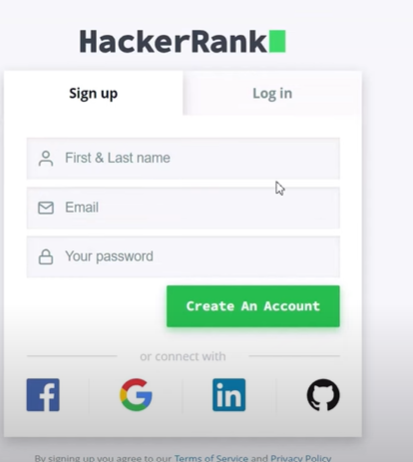
**Detailed OAuth 2.0 Flow**

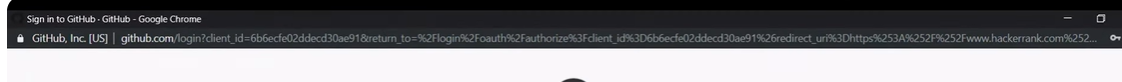
In Simple Way I am a user I want to login to HakerRank website with out registration I can login by using login with gitgub , now when I click login with github it will redirect to github page with clientId and our hackerrank call back url (client id here unique id allocated for hacker rank by github, (call back url used again to come back to hacker rank page) now we get github login when we login github ask to authorize for reading github user name date of birth and password once we authorize it will call back hacker rank , hacker rank get authorization code , now hacker rank will call github Authorization server with the authorization code and clientId for access token , then Authorization server will validate and send access token and refresh token back to hacker rank , then hacker rank will call github resource server to get user name , email, dateofbirth

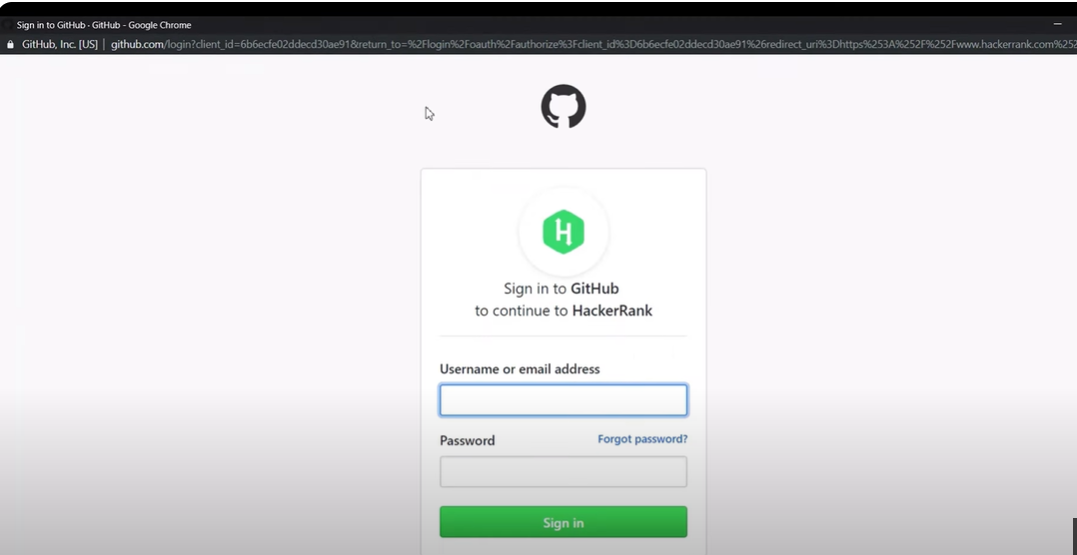
**Client:** Hacker Rank **, Resource Server**: GitHub User Details Service

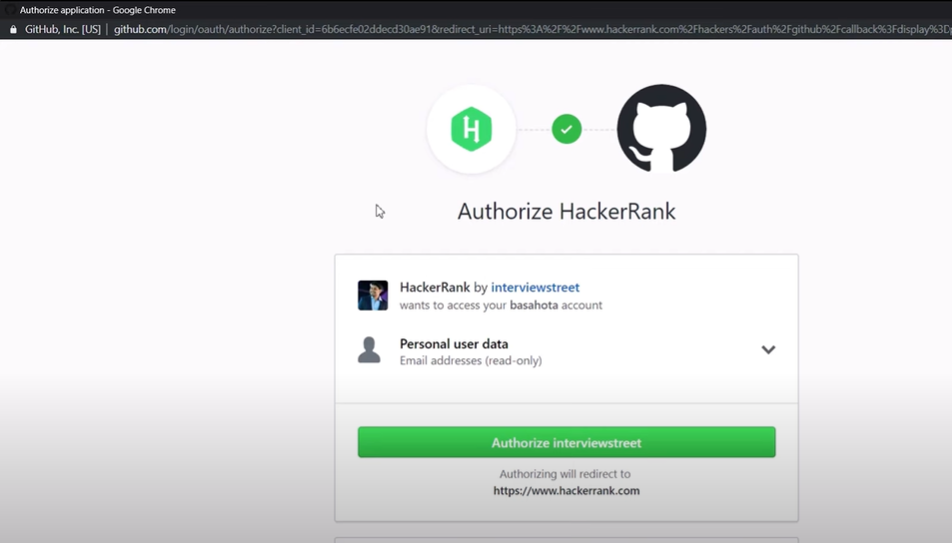
**Authorization Server** : GitHub Authorization Service

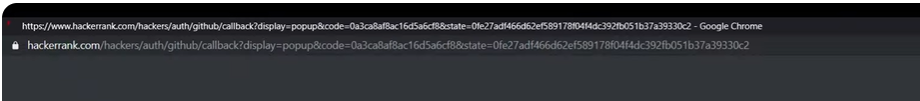
Just Assume Client as any web / mobile any application

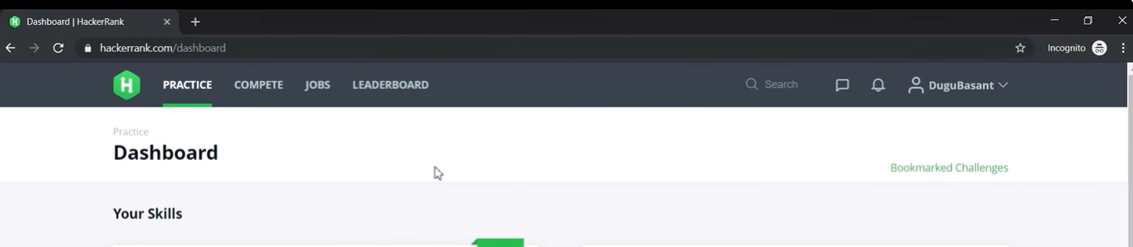












**1. User Requests Resource Access**

* **Action:** The user tries to access a protected resource (e.g., user profile) on a client application (e.g., a web or mobile app).
* **Flow:** If the user is not authenticated, the client application needs to authenticate the user. It redirects the user to the Authorization Server.

**2. Authorization Request**

* **Action:** The client application constructs an authorization request and includes the client\_id in this request. The client\_id is used to identify which client is making the request.
* **Example URL:**

http

Copy code

GET /oauth/authorize

?response\_type=code

&client\_id=my-client-id

&redirect\_uri=https://myapp.com/callback

&scope=read\_profile

* + response\_type=code specifies that the client is requesting an authorization code.
  + client\_id=my-client-id identifies the client application.
  + redirect\_uri=https://myapp.com/callback is the URL where the Authorization Server will redirect the user after authentication.
  + scope=read\_profile specifies the permissions requested by the client.

**3. User Authentication**

* **Action:** The Authorization Server presents a login page where the user provides credentials (username and password). If authentication is successful, the Authorization Server issues an authorization code.

**4. Authorization Code Grant**

* **Redirect with Authorization Code:**

http

Copy code

GET /callback

?code=authorization-code

* **Action:** The user is redirected back to the client application with the authorization code. The client application will now exchange this authorization code for an access token.

**5. Token Exchange**

* **Action:** The client application sends a request to the Authorization Server to exchange the authorization code for an access token. The client\_id and client\_secret (if applicable) are included in this request for authentication.
* **Token Request:**

http

Copy code

POST /oauth/token

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

Authorization: Basic base64(client-id:client-secret)

grant\_type=authorization\_code

&code=authorization-code

&redirect\_uri=https://myapp.com/callback

* + client-id and client-secret are sent as a Basic Authentication header to authenticate the client application.
  + grant\_type=authorization\_code indicates the type of grant being used.
  + code=authorization-code is the authorization code received earlier.
  + redirect\_uri=https://myapp.com/callback should match the redirect URI used in the initial request.
* **Token Response:**

json

Copy code

{

"access\_token": "access-token",

"token\_type": "bearer",

"expires\_in": 3600,

"refresh\_token": "refresh-token"

}

**6. Access Resource with Token**

* **Action:** The client application uses the access token to request resources from the Resource Server.
* **Example Request:**

http

Copy code

GET /user/profile

Authorization: Bearer access-token

* **Resource Server:** Validates the access token (checking its signature and claims) and provides access to the resource if the token is valid.

**7. Token Expiry and Refresh**

* **Action:** When the access token expires, the client application uses the refresh token to request a new access token.
* **Refresh Token Request:**

http

Copy code

POST /oauth/token

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

Authorization: Basic base64(client-id:client-secret)

grant\_type=refresh\_token

&refresh\_token=refresh-token

* **Refresh Token Response:**

json

Copy code

{

"access\_token": "new-access-token",

"token\_type": "bearer",

"expires\_in": 3600,

"refresh\_token": "new-refresh-token"

}

**Summary of the Role of Client ID**

1. **Authorization Request:** The client\_id identifies the client application making the request to the Authorization Server.
2. **Token Exchange:** The client\_id and client\_secret authenticate the client application when exchanging the authorization code for an access token.
3. **Resource Access:** The access token is used to access protected resources, but the client\_id is not included in this part of the flow.

In the OAuth 2.0 token exchange flow, when a client application requests a new access token using a refresh token, the refresh\_token parameter in the request is the refresh token that was originally issued along with the access token.

**Refresh Token Request**

Here’s how it works in detail:

1. **Initial Token Request:** When the user initially logs in and the client exchanges the authorization code for an access token, the Authorization Server also issues a refresh token (if the offline\_access scope is requested or supported).
   * **Example Token Response:**

json

Copy code

{

"access\_token": "access-token",

"token\_type": "bearer",

"expires\_in": 3600,

"refresh\_token": "refresh-token"

}

* + The refresh\_token here is a unique token that can be used to obtain new access tokens after the original access token expires.

1. **Using Refresh Token to Obtain a New Access Token:**
   * **Request to Exchange Refresh Token for a New Access Token:**

http

Copy code

POST /oauth/token

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

Authorization: Basic base64(client-id:client-secret)

grant\_type=refresh\_token

&refresh\_token=refresh-token

* + In this request:
    - grant\_type=refresh\_token specifies that the request is to use the refresh token grant type.
    - refresh\_token=refresh-token is the refresh token value obtained previously.

**Example**

Let’s say the refresh token issued initially is abc123. When you make the request to exchange it for a new access token, it would look like this:

http

Copy code

POST /oauth/token

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

Authorization: Basic base64(client-id:client-secret)

grant\_type=refresh\_token

&refresh\_token=abc123

**How the Refresh Token Value is Used**

* **Security:** The refresh token should be securely stored by the client and sent over HTTPS to protect it from being intercepted.
* **Lifetime:** Refresh tokens generally have a longer lifetime than access tokens, but they are still subject to expiration and revocation.
* **Server Validation:** The Authorization Server will validate the refresh token to ensure it is valid and has not been revoked. If valid, it will issue a new access token (and potentially a new refresh token).

**Summary**

* The refresh\_token parameter in the request should be the exact value of the refresh token issued by the Authorization Server.
* This token is used to request a new access token when the current access token expires.

If you have any specific questions or need further clarification on a particular aspect of OAuth 2.0 or the refresh token flow, feel free to ask!

**OAuth2 code Pseudo code**

**@Configuration @EnableAuthorizationServer public class AuthorizationServerConfig extends AuthorizationServerConfigurerAdapter { @Override public void configure(ClientDetailsServiceConfigurer clients) throws Exception {**

**@Configuration @EnableResourceServer public class ResourceServerConfig extends ResourceServerConfigurerAdapter { @Override public void configure(ResourceServerSecurityConfigurer resources) throws Exception {**

**Authorization vs. Authentication**

* **Authentication:** The process of verifying the identity of a user or system. For example, checking if a username and password match a record in a database.
* **Authorization:** The process of determining what resources or actions an authenticated user or system is allowed to access. It’s about granting permissions and access levels.

**JWT (JSON Web Tokens)**

**Definition:**  
JWT is a compact, URL-safe token format used for securely transmitting information between parties as a JSON object. It is often used for authentication and authorization.

**Structure:**

* **Header:** Contains metadata about the token, including the type (JWT) and signing algorithm (e.g., HS256).
* **Payload:** Contains the claims or statements about the user or application. It can include custom claims and standard claims like iss (issuer), sub (subject), and exp (expiration).
* **Signature:** Ensures the token's integrity and authenticity. It is created by signing the header and payload with a secret key or private key.

**When to Use:**

* When you need a compact, self-contained token for authentication or authorization.
* For stateless authentication where the token itself contains user information and permissions.
* When you want to include custom claims and manage token expiration and validation.

**Advantages:**

* Self-contained and portable, containing all necessary information for authentication.
* Stateless, meaning the server doesn’t need to store session data.
* Supports various signing algorithms for secure transmission.

**JWT Flow in Microservices**

* 1. A client sends a Request to access a Secured Resource then in gateway it will check if the request header contain JWT access token or not. If it doesn’t contain token it will send error response if it contain a JWT token it validates and if its valid then allow to access a requested resource.
  2. If to get access token user will request /getaccesstoken with user name and password then gateway will call identity service, in identity service it will validate credentials.
  3. Upon successful authentication, the service generates a JWT and returns it to the client.

4.The client sends a request with the JWT in the Authorization header to the API Gateway.

5. The API Gateway validates the JWT (checks the signature, expiration, and claims).

6. If the token is valid, the gateway forwards the request to the appropriate microservice.

7.If the token is invalid, the gateway returns an unauthorized response to the client.

@Component

**public** **class** AuthenticationFilter **extends** **AbstractGatewayFilterFactory**<AuthenticationFilter.Config> {

@Override

**public** GatewayFilter apply(Config config) {

check if request url is allowed or not (check with our list of allowed urls, match with request url if it matches then allow ) or else if url doesn’t match then check for request headers contain bearer token or not then if token not present return 401, if we have access token the validate token structure with base64 decode , then validate signature with help of header used algorithm then payload part for claims like issuer , issue date, expiry date , subject etc…