JWT Structure:

1. The JWT consists of three parts:

    a. A (Header): Typically specifies the signing algorithm (like RSA) and the type (JWT).

    b. B (Payload): Contains the claims data (like client_id, authorization_code, etc.,).

    c. C (Signature): Created by signing the encoded header and payload using a secret key.

2. Base64 Encoding:

    * Both the header and the payload are base64 URL encoded, not just base 64 encoded. This ensures that the output can be safely

    transmitted in URLs.

3. Signature Creation:

    * For RSA, you typically sign the combination of the encoded header and encoded payload using the RSA secret key. This process looks like this:

        a. Create a string: A + "." + B.

        b. Sign this string using the RSA secret key to produce the signature.

4. Client Secret:

    Used for digital signature generation.

5. Final JWT Token:

    * The final JWT token is structured as: A + '.' + B + '.' + C where C is the RSA-signed result.

6. Security:

    * The signature allows the recipient to verify that the token has not been altered and is authentic. Only someone with the private key

    (the authorization server) can create a valid signature.

Asymmetric Signing: Involves a pair of keys- a private key for signing and a public key for verification. This allows the server to

sign the JWT without needing to share the private key.

Symmetric Signing: Uses a single shared secret key for both signing and verification. While this is simpler, it means both parties must keep

the shared secret key secure.

In our user service, the algorithm used for JWT token generation is 'RS256' which is a type of Asymmetric algorithm.

1. Asymmetric Algorithm:

* When using an asymmetric algorithm (like RSA), there are indeed two keys: a private key and a public key.

2. Key Ownership:

* The authorization server knows the private key and uses it to sign the JWT. The private key is kept secret and secure.

* The public key can be shared with other parties (like resource server or clients) to validate the token.

3. Token Generation:

* The authorization server generates the JWT token by creating a digital signature using the private key. The signature allows anyone with the corresponding

public key to verify the authenticity of the token.

4. Validation by Resource Server:

* The resource server (or any client) can use the public key shared by the authorization server to validate the signature of the JWT.

If the signature is valid, it confirms that the token was indeed signed by the authorization server and has not been tampered with.

Summary:

1. The authorization server knows both the private and public keys.

2. The private key is used for signing the JWT.

3. The public key is shared with the resource server (or other clients) for token validation.

1. JWT Structure: A JWT consists of three parts: Header, Payload, and Signature. The signature is created using the private key and servers to validate

the integrity of the token.

2. Verification Without Constant Communication:

* Public Key Usage: The resource server can store the public key received from the authorization server. Once it has the public key, the resource server

can validate incoming JWTs without needing to contact the authorization server each time.

* This is what gives JWTs their self-validating feature: as long as the public key is known and stored securely, the resource server can independently

verify the signature on the token.

3. Key Retrieval:

* The public key can be retrieved from the authorization server during an initial setup or configuration phase. It can also be made available through

a secure endpoint (like JWKS endpoint), where the resource server can periodically check for updates or changes to the public key.

Key Management and Updates:

1. Key Rotation: In environments where keys may change, the resource server can implement a mechanism to periodically retrieve or refresh the public key.

This can be done on a schedule or whenever a token validation fails due to a signature mismatch.

2. Caching: The resource server can cache the public key and only refresh it when necessary, reducing the need for constant communication with the

authorization server.

3. The public key can be retrieved once and stored securely, with mechanisms in place for updating if needed.

JWT Token Generation By The Authorization Server:

JWT Token = A . B . C

A = base64Encode(Header);

B = base64Encode(Payload);

C = RS256(A + '.' + B, private key);

JWT Token Validation By The Resource Server:

[A, B, C] = split(JWT_Token, '.');

if (RS256(A + '.' + B, public key) == C) return true; // represent valid JWT token.

else return false; // represent invalid JWT token.

Configuration used in the resource server application.properties file:

spring.security.oauth2.resourceserver.jwt.issuer-uri=http://localhost:4141

The above line is a configuration property commonly found in a spring application that uses spring security to secure resources via OAuth 2.0 and JWT

(JSON Web Tokens).

1. Property Name:

spring.security.oauth2.resourceserver.jwt.issuer-uri

2. Purpose:

* This property specifies the issuer URI of the JWTs that the resource server expects to receive.

* The issuer URI is part of the JWT claims and typically represents the authorization server that issued the token.

3. Functionality:

* When the resource server receives a JWT, it will validate the token. Part of this validation process includes checking the iss (issuer) claim in

token against the URI specified in this property.

* By defining the issuer-uri, you ensure that the resource server only accepts tokens that have been issued by a trusted authorization server.

4. Security:

* This helps protect against token forgery by verifying that incoming tokens originate from a legtimate source.

* If the iss claim in the JWT does not match the specified issuer-uri, the resource server can reject the token, preventing unauthorized access.

http://localhost:4141, it indicates that the resource server expects tokens issued by an authorization server running locally on port 4141.

spring.security.oauth2.resourceserver.jwt.issuer-uri property is crucial for securing your resource server by ensuring that it only accepts JWTs

from a specified, trusted authorization server. This enhances the security of your application by validating the source of incoming tokens.