Confidential Client Authentication

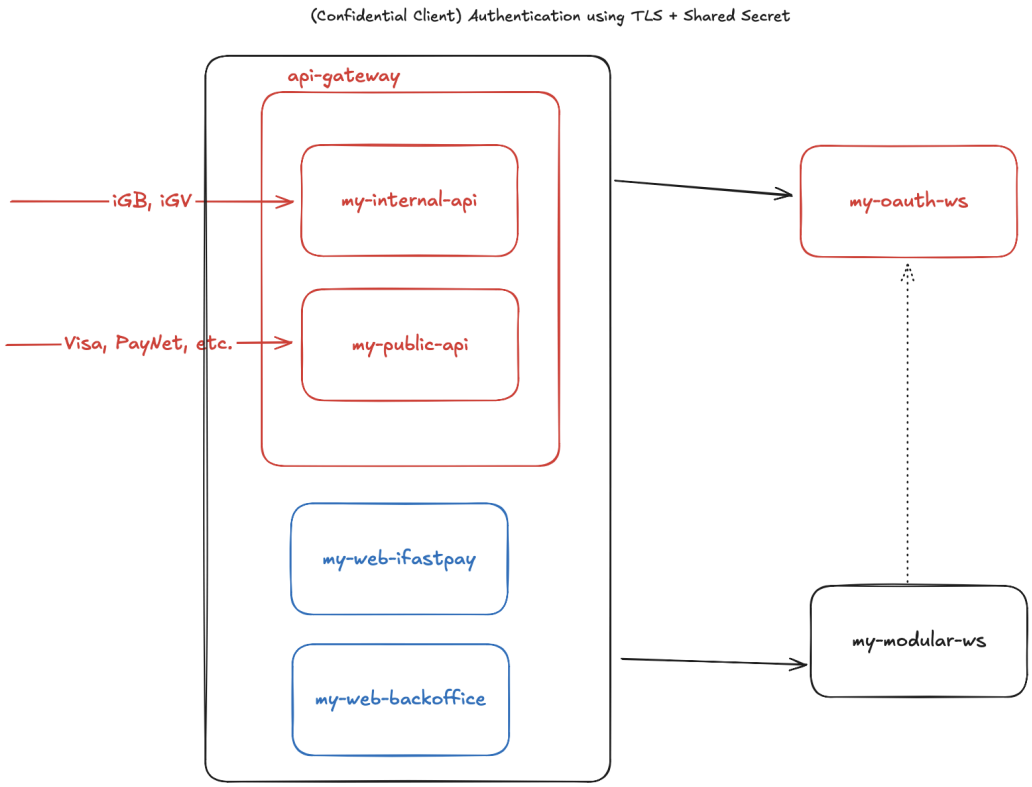
1. Overview (Refer to the *Updated* section)

As defined by OAuth2.0, ***confidential clients*** are clients which have the ability to maintain the confidentiality of the `client_secret`. Typically these clients are only applications that run on a server under the control of the developer, where the source code is not accessible to users.

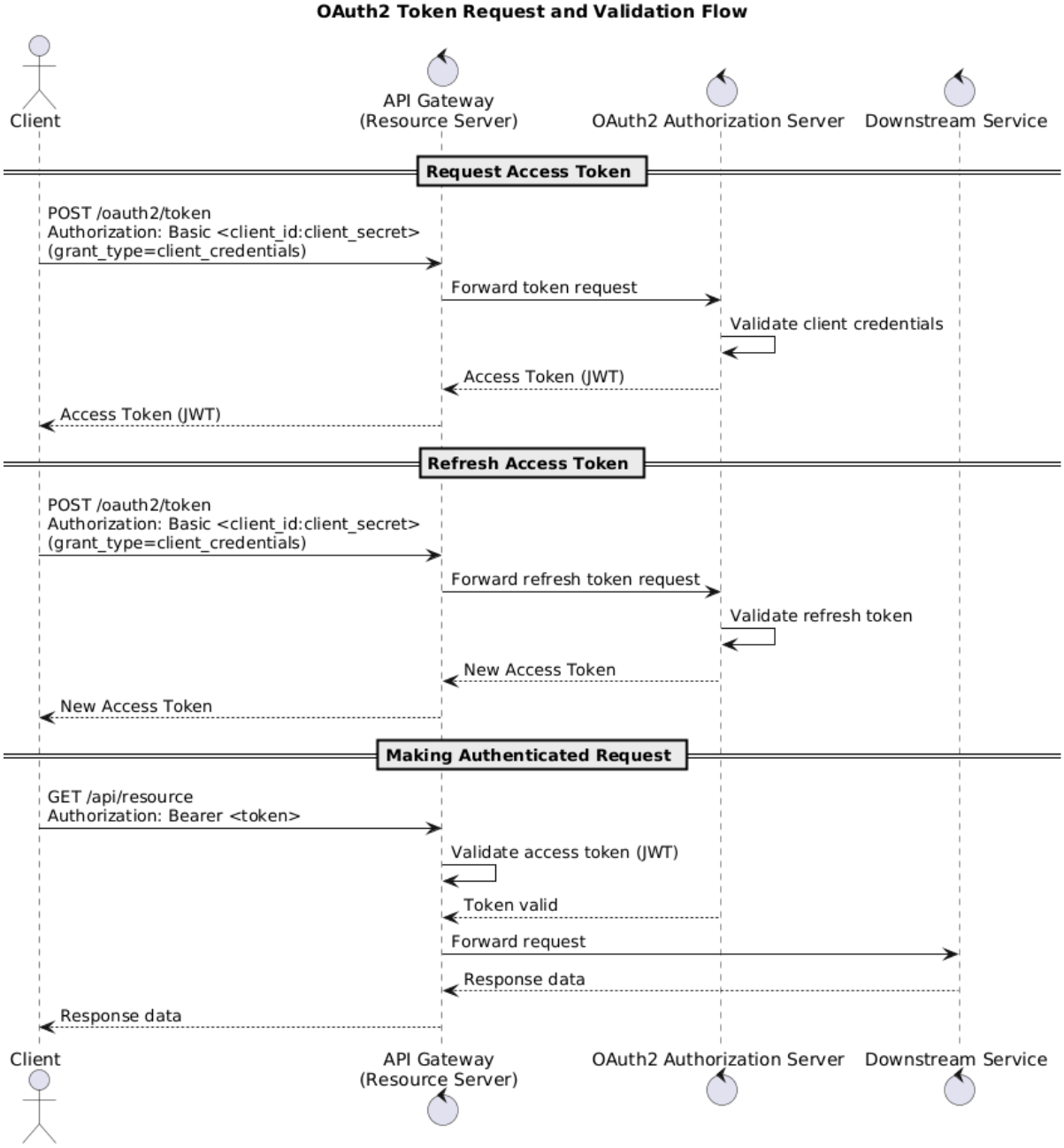
For iFastPay, confidential clients are categorized into two groups, each using different authentication measures.

Confidential Client Type	Justification
Internal Services	**Definition** Internal services are services that reside in the same deployment network as the iFastPay's OAuth2.0 Authorizati Server. **Authentication Method** ● Application Level: OAuth2.0 (Shared Secret) ● **Network Level: TLS/ mTLS** **Justification** Since confidential clients can securely store shared secrets, is safe to authenticate them using client credentials over TL
External Services	**Definition** External services are services that doesn't reside in the sam deployment network as the iFastPay's OAuth2.0 Authorizati Server. E.g. iGB, iGV, Visa, Paynet, etc. **Authentication Method** ● Application Level: OAuth2.0 (Shared Secret) ● **Network Level: TLS/ mTLS + IP Filtering + Intranet**

	Justification Since confidential clients can securely store shared secrets, is safe to authenticate them using client credentials over TL



- As illustrated in the system architecture, the API Gateway acts as a reverse proxy for the `my-modular-ws` module, serving both internal and external clients.
- For simplicity, `my-internal-api` and `my-public-api` are collectively referred to as the API Gateway in this documentation.
- The Gateway serves as the central point for authentication and token validation. The *OAuth2 Authorization Server* and *Resource Server* functionalities are handled by the Gateway and Authorization Server components respectively.



1. Request Access Token
    a. The *OAuth2.0 Client* requests an access token from the `POST /oauth2/token` endpoint. The client must include a ***Basic Authorization header*** (`Authorization: Basic <base64(client_id:client_secret)>`) for authentication.
    b. The API Gateway forwards this request to the *OAuth2 Authorization Server*.
    c. The *Authorization Server* validates the client credentials stored in its database.
    d. If the credentials are valid, it issues an access token and returns it to the Gateway,

which then forwards the response back to the client.

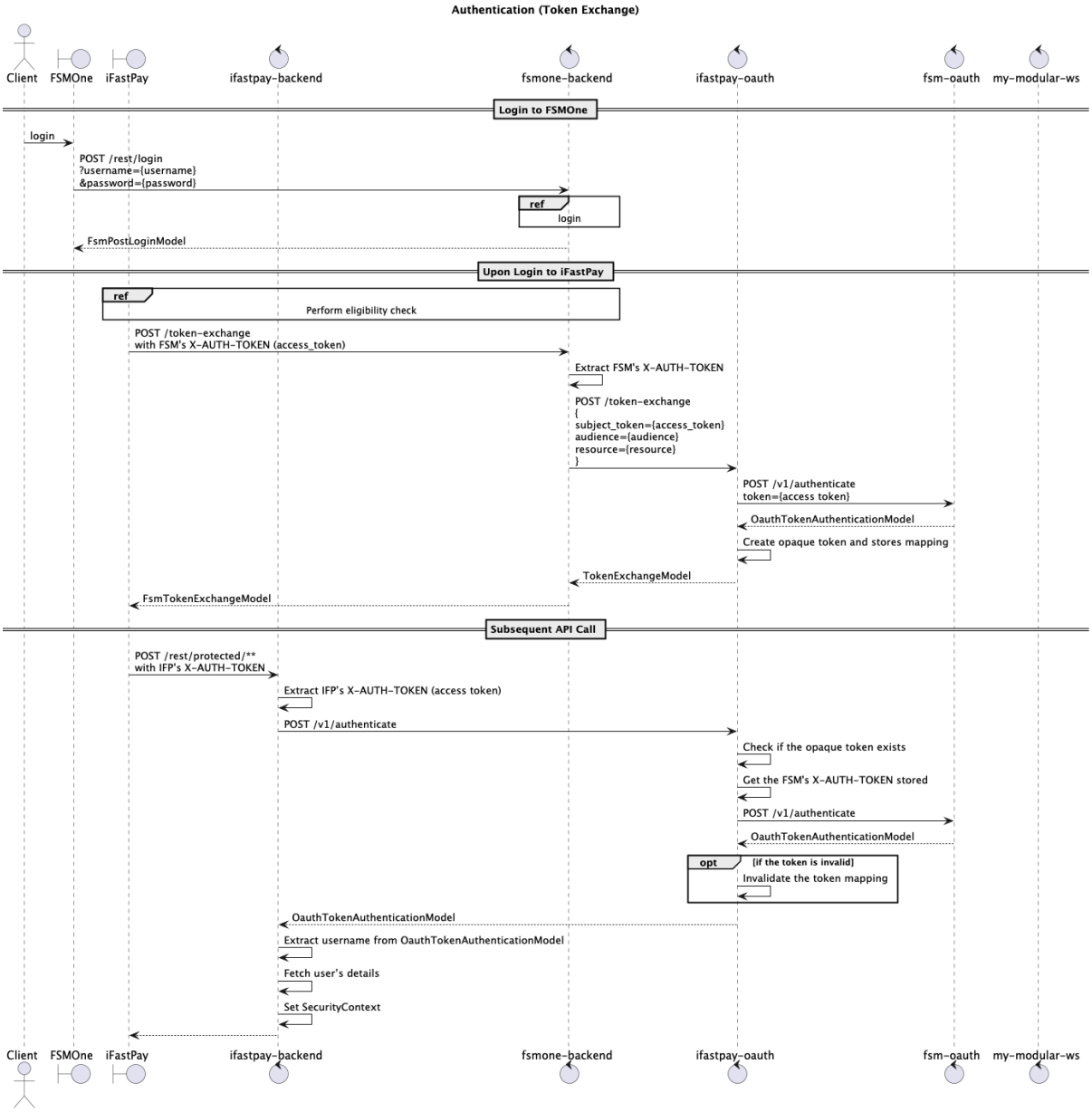
2. **Refresh access token**
    a. Same as requesting a new access token

3. **Making Requests**
    a. Whenever the *OAuth2.0 Client* calls a protected API, it must include the ***Bearer token*** in the Authorization header.
    b. The API Gateway acts as the *OAuth2 Resource Server* to validate the token's authenticity.
    c. If the token is valid, the Gateway forwards the request to the appropriate downstream service.
    d. Otherwise, it rejects the request.

Summary

- Since no user login is involved, OAuth2 Clients should request a new access token when needed instead of using a refresh token.
- The API Gateway will decode and validate JWT tokens issued by the Authorization Server.
- To reduce development complexity, application-level security will be minimal. The system relies primarily on TLS and network-level restrictions for protection.

Updated



In the updated design, FSMOne will first send a request to the iFastPay OAuth2 Authorization Server to obtain an opaque token that is used specifically for accessing iFastPay services.

The iFastPay Authorization Server will generate this *opaque token* (which is simply a random string used for lookup rather than a self-contained JWT) and store a mapping between the opaque token and FSMOne's access token.

Each token mapping record may include fields such as:
aud, res, subject_token, iss, scope, and other necessary attributes.

When an API request with the opaque token arrives, the iFastPay Authorization Server will:

1. Look up the corresponding record in the token mapping table.
2. Retrieve the original FSMOne access token.
3. Use the FSMOne access token to validate whether the user's session is still valid via the FSMOne authentication endpoint.

This design acts as a **hybrid model** between shared token usage and token exchange. It reuses the original FSMOne access token for validation but introduces a new opaque token layer, giving users the impression of distinct tokens within the iFastPay ecosystem. Extra security mechanism can be implemented through this layer as well.

Technical consideration
This approach provides a foundation for centralized token management, but the architecture must be designed carefully to ensure future compatibility and integration flexibility. In particular, database schema and component boundaries should be planned with backward compatibility and scalability in mind.

The proposed database structure includes two schemas:

- **Token Mapping Schema** – stores relationships between iFastPay opaque tokens and source system tokens (e.g., FSMOne).
- **Introspection Endpoint Schema** – stores information about target authorization servers and their corresponding introspection endpoints for validating tokens.

When a request arrives:

1. iFastPay first checks the token mapping table to identify which authorization server the token belongs to.
2. It then consults the introspection endpoint table to determine which endpoint to call.
3. Finally, it invokes that endpoint to validate the token's status.

To achieve maintainability and adherence to the Open/Closed Principle, the system should employ a Strategy Pattern combined with Dependency Injection:

- Each target authorization server (e.g., FSMOne, future systems) will have its own strategy implementation for token introspection or validation. (instead of using reflection which is implicit or if-else which violates OCP)
- A **Simple Factory** can be used to instantiate the correct strategy at runtime based on the token mapping.
- A **centralized iFastPay-centric Token Model (defined via an interface)** will act as an adapter layer, ensuring a unified abstraction across various token types and external authentication formats.'

This hybrid model enables easy extension for future systems while maintaining clear separation of concerns.

While RFC 8693 (OAuth2 Token Exchange) remains the long-term standard for interoperability, the current custom approach serves as a practical interim solution, especially since FSMOne's OAuth implementation might not be fully compatible with the RFC 8693 specification.

In essence, we can either use the Spring Authorization Server (SAS) token exchange feature together with a custom adapter between the two systems, or implement a simplified token exchange mechanism (such as the previously described token mapping approach). Regardless of which approach is chosen, an adapter will still be required. By combining the Strategy Pattern with a Simple Factory, we can keep the codebase modular, maintainable, and easily extendable for future integration needs.

2. Software Bill of Materials (SBOM)

Bill of Materials	Version
`org.springframework.boot:spring-boot-dependencies`	3.4.3
`org.springframework.cloud:spring-cloud-dependencies`	2024.0.1

Reference (If it doesn't render properly in Safari, consider to use Google instead.)

- https://repo1.maven.org/maven2/org/springframework/boot/spring-boot-dependencies/3.4.3/spring-boot-dependencies-3.4.3.pom
- https://repo1.maven.org/maven2/org/springframework/cloud/spring-cloud-dependencies/2024.0.1/spring-cloud-dependencies-2024.0.1.pom

Dependencies	Version
`org.springframework.cloud:spring-cloud-starter-gateway`	4.2.1
`org.springframework.boot:spring-boot-starter-security`	6.4.3
`org.springframework.boot:spring-boot-starter-oauth2-authorization-server`	3.4.3
`org.springframework.boot:spring-boot-starter-oauth2-client`	3.4.3
`org.springframework.boot:spring-boot-starter-oauth2-resource-server`	3.4.3

3. OAuth2.0 Endpoints

3.1. Endpoints exposed

URL (default)	RFC / Spec	Description
/oauth2/token	RFC 6749	Exchanges credentials for tokens.Issues access, refresh, ID tokens.
/oauth2/introspect	RFC 7662	RFC 7662 endpoint for resource servers to validate opaque tokens (active, scopes, subject, etc.). Requires client auth.
/.well-known /oauth-authorization-server	RFC 8414	Machine-readable discovery document for OAuth endpoints & issuer. (Home)
/oauth2/jwks	RFC 7517	Publishes public keys used to verify JWTs issued by the AS. *(enabled when a JWKSource bean is present)*. (Home)

3.2. Endpoints Restricted

URL (default)	RFC	Description
/oauth2/revoke	RFC 7009	RFC 7009 endpoint for clients to revoke access or refresh tokens.
/oauth2/authorize	RFC 6749 §3.1	Starts user authorization (e.g., Authorization Code, PKCE). (Home)
/oauth2/par	RFC 9126	Client pushes the authorization request to AS and gets a request_uri to use at /oauth2/authorize. (Home)
/oauth2/device_authorization	RFC 8628	Starts device flow on constrained devices; returns device_code and user_code. (Home)
/oauth2/device_verification	RFC 8628	User enters user_code here to approve the device. (Home)

/.well-known/openid-configuration	OIDC Discovery 1.0	OIDC provider metadata (lists userinfo, end_session_endpoint, etc.). (Home)
/userinfo	OIDC Core §5.3	Returns claims about the authenticated end-user (requires JWT decoder). (Home)
/connect/logout	OIDC RP-Initiated Logout	Ends the RP session per OIDC logout spec. (Home)
/connect/register	OIDC Dynamic Client Registration 1.0	Allows RPs to register/read clients dynamically when enabled. (Home)

These endpoints are not exposed to reduce the attack surface and these endpoints have little use for now.

4. Client Registration

Fields	Value
client-id	Convention: • `public-X`: For third parties access through `my-public-api` • `internal-X`: For third parties access through `my-internal-api`
client-secret	• Generation Algorithm: CSPRNG with AES • Encryption Algorithm: Bcrypt
client-authentication-methods	client_secret_basic
authorization-grant-type	client_credentials
scopes	NIL
client	Refer to Client Settings
token	Refer to Token Settings

4.1. Client Settings

Less Important		
Require Proof Key	false	*Spring Security Default*.
Require Authorization Consent	false	*Spring Security Default*.
Refresh Token TTL	60 minutes	*Spring Security Default*.

4.2. Token Settings

Important		
Access Token TTL	5 minutes	*Spring Security Default*.
Access Token Format	OAuth2TokenFormat. SELF_CONTAINED	*Spring Security Default*.

Token Signature Algorithm	RS256	*Spring Security Default*.
Reuse Refresh Tokens	true	*Spring Security Default*.
Use X509 Certificate Bound Access Token	false	*Spring Security Default*.
Less Important		
Authorization Code TTL	5 minutes	*Spring Security Default*.
Device Code TTL	5 minutes	*Spring Security Default*.
Refresh Token TTL	60 minutes	*Spring Security Default*.

5. Security Algorithm

5.1. Key Generation

For: Client Secret Generation

Algorithm: CSPRNG (Cryptographically Secure Pseudo-Random Number Generator) with AES-256 (Advanced Encryption Standard with Key Size of 256 bit).

```
public static SecretKey generateSecretKey()
        throws NoSuchAlgorithmException {
        KeyGenerator keyGenerator =
KeyGenerator.getInstance("AES");
        keyGenerator.init(256);
        return keyGenerator.generateKey();
}
```

5.2. Key Pair Generation

For: Public-Private Key Pair Generation, e.g. JWK

Algorithm: RSA, Key Size of 3072 bit, public exponent of 65536, CSRPNG (use SecureRandom internally)

```
public static KeyPair generateKeyPair()
        throws NoSuchAlgorithmException,
InvalidAlgorithmParameterException {
        KeyPairGenerator keyPairGenerator =
KeyPairGenerator.getInstance("RSA");
        keyPairGenerator.initialize(new
RSAKeyGenParameterSpec(3072, RSAKeyGenParameterSpec.F4));
        return keyPairGenerator.generateKeyPair();
}
```

5.3. Encryption

For: Encryption/ Decryption

The encrypted JWK uses **AES-GCM** with the following parameters and key derivation setup:

- **Transformation:** AES/GCM/NoPadding
- **Authentication Tag:** 128-bit GCM tag
- **Initialization Vector (IV):** 12 bytes (generated using CSPRNG)
- **Salt:** 16 bytes (generated using CSPRNG)
- **Key Derivation Function (KDF):** PBKDF2WithHmacSHA256
  - **Password Source:** The provided private key (converted to a character array)
  - **Salt:** 16-byte random salt
  - **Iteration Count:** 210,000
  - **Derived Key Length:** 256 bits
- **Secret Key:** Derived from the hashed private key using SHA-256
- **Cipher Mode:** AES in GCM mode for authenticated encryption
- **Additional Authenticated Data (AAD):** Concatenation of salt and IV (28 bytes total)
- **Output Structure:**
  [16-byte salt][12-byte IV][ciphertext with 128-bit GCM tag]
- **Encoding:** The final byte sequence is Base64-encoded for storage or transmission.

```
public static String encrypt(String plainText, String privateK
throws Exception {
    // 1. Define SecretKey
    byte[] salt = new byte[16];
    new SecureRandom().nextBytes(salt);
    PBEKeySpec pbeKeySpec = new PBEKeySpec(password, salt,
210000, 256);
    SecretKeyFactory factory =
SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
    byte[] keyBytes =
```

```
factory.generateSecret(keySpec).getEncoded();
    SecretKey secretKey = new SecretKeySpec(keyBytes, cipher);
    pbeKeySpec.clearPassword();

    // 2. Define GCMParameterSpec
    byte[] iv = new byte[12];
    new SecureRandom().nextBytes(iv);
    GCMParameterSpec gcmParameterSpec = new GCMParameterSpec(1
iv);

    // 3. Define Cipher
    Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
    cipher.init(Cipher.ENCRYPT_MODE, secretKeySpec,
gcmParameterSpec);
    byte[] aad = ByteBuffer.allocate(16 + 12).put(salt)
        .put(iv).array();
    cipher.updateAAD(aad);

    // 4. Perform encryption
    byte[] plainTextBytes = plainText.getBytes(DEFAULT_CHARSET
    byte[] cipherTextBytes = cipher.doFinal(plainTextBytes);
    byte[] outputBytes = ByteBuffer
        .allocate(16 + 12 + cipherTextBytes.length)
        .put(salt).put(iv).put(cipherTextBytes).array();

    // 5. Return result
    return Base64.getEncoder().encodeToString(outputBytes);
}
```

5.4. JWK Management

JWK Generation: Refer to Section 5.2.

JWK Encryption: Refer to Section 5.3.

6. Setup

6.1. Gateway

```
    @Bean
    SecurityWebFilterChain
defaultSecurityWebFilterChain(ServerHttpSecurity http) {
       return http
            .csrf(ServerHttpSecurity.CsrfSpec::disable)
            .httpBasic(ServerHttpSecurity.HttpBasicSpec::disable)
            .formLogin(ServerHttpSecurity.FormLoginSpec::disable)
            .authorizeExchange(ex -> ex
                .pathMatchers("/oauth2/token", "/.well-
known/**").permitAll()
                .anyExchange().authenticated()
            )
            .oauth2ResourceServer(o -> o.jwt(Customizer.withDefaults()))
            .build();
    }
```

```
 security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:9000
      client:
        provider:
          local-as:
            issuer-uri: http://localhost:9000
        registration:
          api-gateway:
            provider: local-as
            client-id: api-gateway
            client-secret: "secret"
            authorization-grant-type: client_credentials
            client-authentication-method: client_secret_basic
```

6.2. Authorisation Server

my-ifast-pay-backend/my-oauth2-ws

```
@Bean
    public SecurityFilterChain oauthAuthServerSfc(
        HttpSecurity http,
        FilterChainExceptionHandlerFilter filterChainExceptionHandlerFilter,
        OAuthAccessDefinedHandler oAuthAccessDefinedHandler,
        OAuthAuthorizationDeniedExceptionHandler
oAuthAuthorizationDeniedExceptionHandler,
        RegisteredClientRepository registeredClientRepository
    ) throws Exception {
        OAuth2AuthorizationServerConfigurer authServerConfigurer =
OAuth2AuthorizationServerConfigurer.authorizationServer();
       http
            .securityMatcher(authServerConfigurer.getEndpointsMatcher())
            .with(authServerConfigurer, authServer -> authServer

.authorizationServerSettings(AuthorizationServerSettings.builder().build())
                .registeredClientRepository(registeredClientRepository)
            )
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers(
                    "/oauth2/device_authorization",
                    "/oauth2/device_verification"
                ).denyAll()
                .anyRequest().authenticated()
            )
            .csrf(AbstractHttpConfigurer::disable)
            .exceptionHandling(ex -> ex

.authenticationEntryPoint(oAuthAuthorizationDeniedExceptionHandler)
                .accessDeniedHandler(oAuthAccessDefinedHandler)
            )
            .formLogin(AbstractHttpConfigurer::disable)
        return http.build();
    }
```

```
INSERT INTO oauth2_registered_client (
    id,
    client_id,
    client_id_issued_at,
    client_secret,
    client_secret_expires_at,
    client_name,
    client_authentication_methods,
    authorization_grant_types,
    redirect_uris,
    post_logout_redirect_uris,
    scopes,
    client_settings,
    token_settings
)
VALUES (
           '9ab9c0f4-2eef-4807-9072-0bc1e9925f1f',
           'api-postman',
           CURRENT_TIMESTAMP,
           '{noop}secret',
           null,
           'api-postman',
           'client_secret_basic',
           'client_credentials',
           null,
           null,
           'api.write,api.read',
           '{
         "@class" : "java.util.Collections$UnmodifiableMap",
         "settings.client.require-proof-key" : false,
         "settings.client.require-authorization-consent" : false
       }',
           '{
         "@class" : "java.util.Collections$UnmodifiableMap",
         "settings.token.reuse-refresh-tokens" : true,
         "settings.token.x509-certificate-bound-access-tokens" : false,
         "settings.token.id-token-signature-algorithm" : [
"org.springframework.security.oauth2.jose.jws.SignatureAlgorithm", "RS256" ],
         "settings.token.access-token-time-to-live" : [ "java.time.Duration",
300.000000000 ],
         "settings.token.access-token-format" : {
           "@class" :
"org.springframework.security.oauth2.server.authorization.settings.OAuth2TokenFor
mat",
           "value" : "self-contained"
```

```
        },
        "settings.token.refresh-token-time-to-live" : [ "java.time.Duration",
3600.000000000 ],
        "settings.token.authorization-code-time-to-live" : [
"java.time.Duration", 300.000000000 ],
        "settings.token.device-code-time-to-live" : [ "java.time.Duration",
300.000000000 ]
      }'
      );
```

The SQL code is completely auto-generated.