International Chamber of Commerce
33-43 avenue du Président Wilson, 75116 Paris, France
+33 (0) 1 49 53 28 28 | iccwbo.org

ICC Banking Commission

Working group on APIs development Briefing Paper n°1

Subject: Security protocol recommendation for trade finance products APIs

29 August 2023

Overview

This proposal is created under ICC led Trade Finance APIs discussion forum, on the technical security aspect for Trade Finance APIs.

It covers the security topic in six (6) parts:

1. Onboarding
2. Authentication
3. Authorization
4. Identity Assertion
5. Customer Consent
6. Payload encryption (API Request/Response)

The intention is to cover how Trade Finance APIs should operate within the Trade community, rather than generic and pure API security technology.

To describe how API calls run securely within Trade Finance community, the following terms are used in the proposal:

Entity

This refers to the organization that send/receive Trade Finance APIs, including FI/Banks, Corporates, FinTechs, and potentially NBFI.

They are generically referred as “entity” in this technical proposal, although playing different roles in Trade Finance itself.

Platform

This refers to the API gateway included system components in above defined “entity” organization, for example:

Corporates system often referred as ERP, with its API infrastructure;

FinTech often provide SaaS platform, equipped with API capabilities;

FIs have banking systems, with its API infrastructure.

They are generically referred as “platform” in this proposal that send/receive Trade Finance APIs.

Customer

This refers to the corporate customer of one or more Entities. This is used in this document to describe an Entity transacting with another Entity that is acting on-behalf-of a Customer.

Customer or Customer User

This refers to the corporate customer end user.

Onboarding

There two types of onboarding:

1. Direct Onboarding (B2B)
   This is the primary onboarding between two Entities. This can include a public key exchange and/or a digital certificate request.
2. Indirect Onboarding (B2B2B)
   This is the onboarding of a customer (and also an end user of the customer) that will interact with one Entity (Partner) that will act on-behalf-of the Customer to another Entity (Bank). The pre-requisite of this onboarding is still the Direct Onboarding between the two Entities and the onboarding of the customer to both Entities.

Direct Onboarding

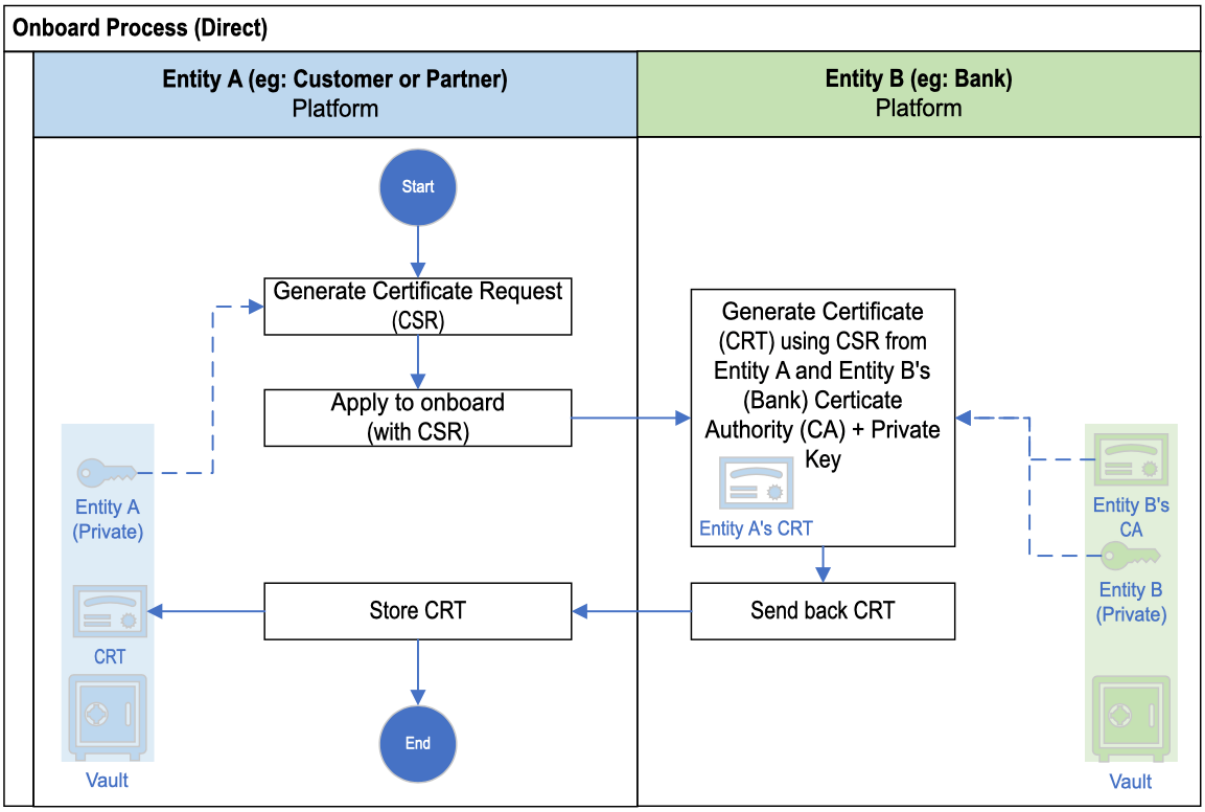
Assume that Entity A wishes to consume Entity B's (Bank) Trade Finance APIs.

mTLS Onboarding

mTLS Onboarding requires a Certificate Request (CSR; generated from Entity A's private key) to be sent to Entity B, who will then issue a Certificate (CRT; generated from Entity B's Private Key and Entity B's Certificate Authority (CA)).

When an mTLS connection is made, both sides present their Certificate during the handshake and both sides can verify the authenticity of the Certificate, thus providing Authentication.

Process Flow (mTLS Onboarding)



Process Steps (mTLS Onboarding)

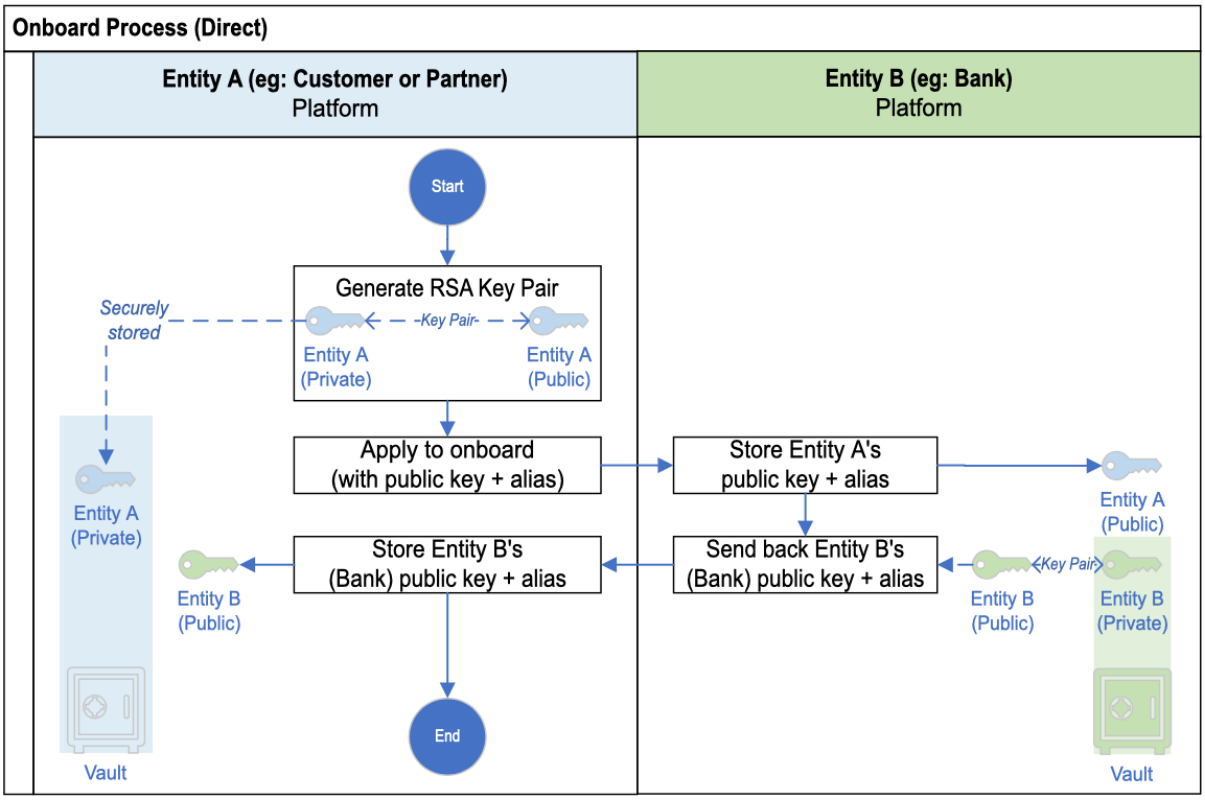
1. Entity A generates a Certificate Request (CSR) using Entity A's new or existing RSA Private Key (used for mTLS)
2. Entity A applies to onboard to Entity B (Bank) by providing Entity A's CSR.
3. Entity B (Bank) generates a Signed Certificate (CRT) using: (i) Entity B's (Bank) root or intermediate Certificate Authority (CA); (ii) Entity B's Private Key; and (iii) Entity A's Certificate Request (CSR).
4. Entity B (Bank) passes back the Signed Certificate (CRT)
5. Entity A stores the Signed Certificate (CRT). It is not mandatory to store the CRT in a Vault.

Key Exchange Onboarding

Key Exchange Onboarding requires both Entities to share their Public Key with each other so that two-way digital signing and encryption can be performed.

When mTLS isn't used only Entity B (Bank) presents a Certificate, proving its authenticity. Therefore in order for Entity A to prove its authenticity, Entity A must digitally sign a JWT using Entity A's Private Key. Entity B can then verify the digital signature using Entity A's Public Key, thus providing Authentication.

Process Flow (Key Exchange Onboarding)



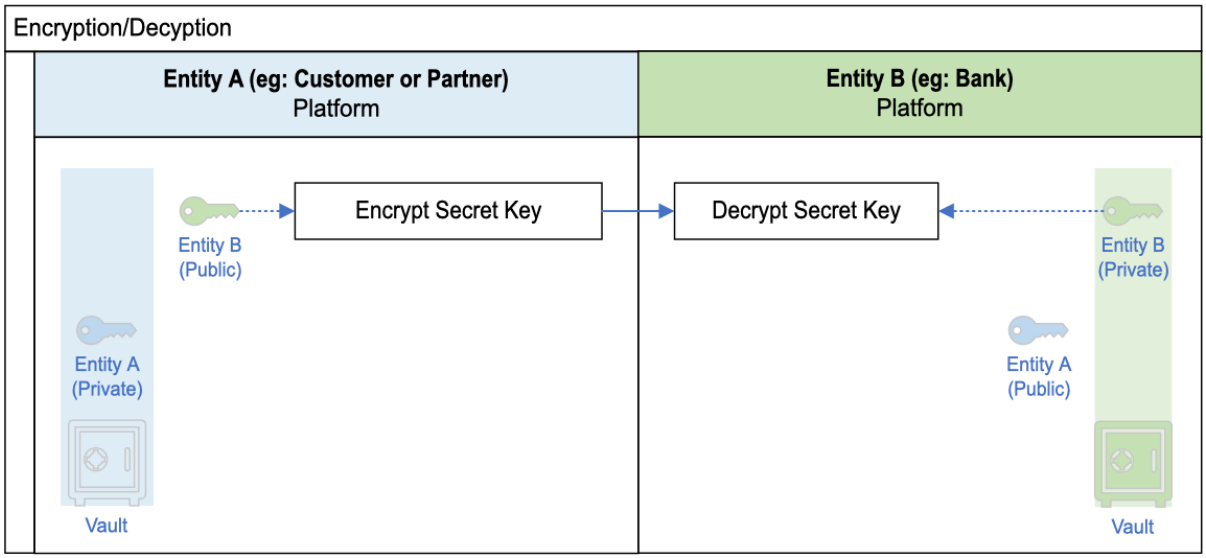
Process Steps (Key Exchange Onboarding)

1. Entity A generates an RSA Key Pair, storing the Private Key in a secure Vault.
   Optionally an existing RSA Key Pair can be used.
2. Entity A applies to onboard to Entity B (Bank) by providing Entity A's Public Key and an alias to uniquely identify the Public Key.
3. Entity B (Bank) stores Entity A's Public Key.
   It is not necessary to store Public Keys in a Vault.
4. Entity B (Bank) passes back Entity B's (Bank) Public Key.
   Optionally a new set of RSA Key Pairs could be generated for each Entity onboarded.
5. Entity A stores Entity B's (Bank) Public Key.
   It is not necessary to store Public Keys in a Vault.

Key Pair Usage

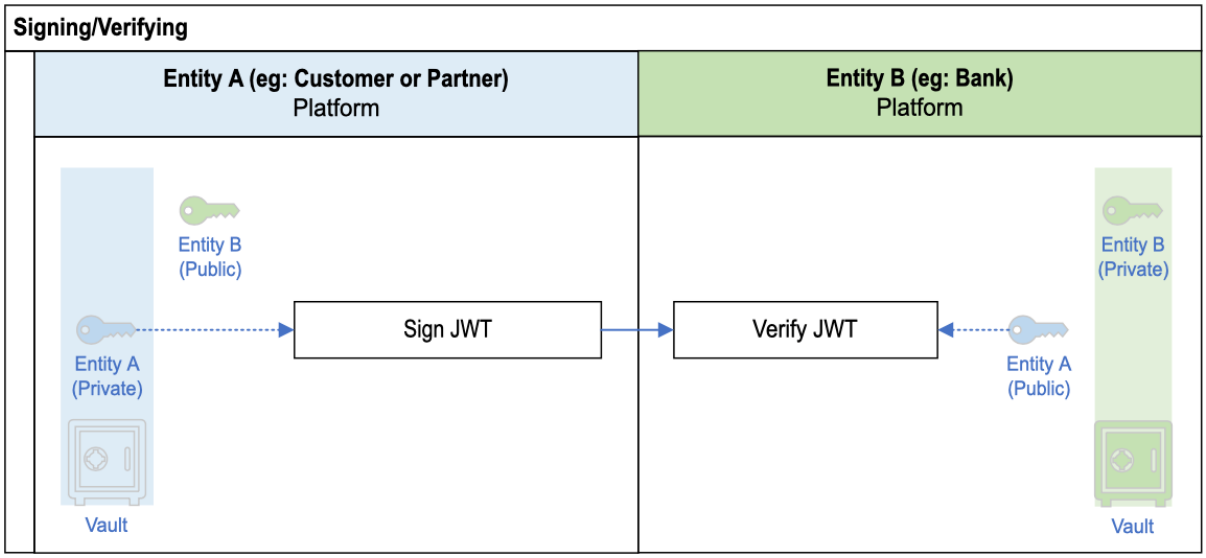
Encryption/Decryption

A recipient's Public Key is used to encrypt, so that only the recipient can decrypt using the recipient's Private Key. This is primarily used to encrypt the Secret Key that is used to encrypt the message (and the synchronous response).



Signing/Verifying

A sender's Private Key is used to digitally sign, so that the recipient can use the sender's Public Key to verify the signature is authentic. This is a form of Authentication, but also the digital signature verifies that the claims in a JWT are unaltered in transit.



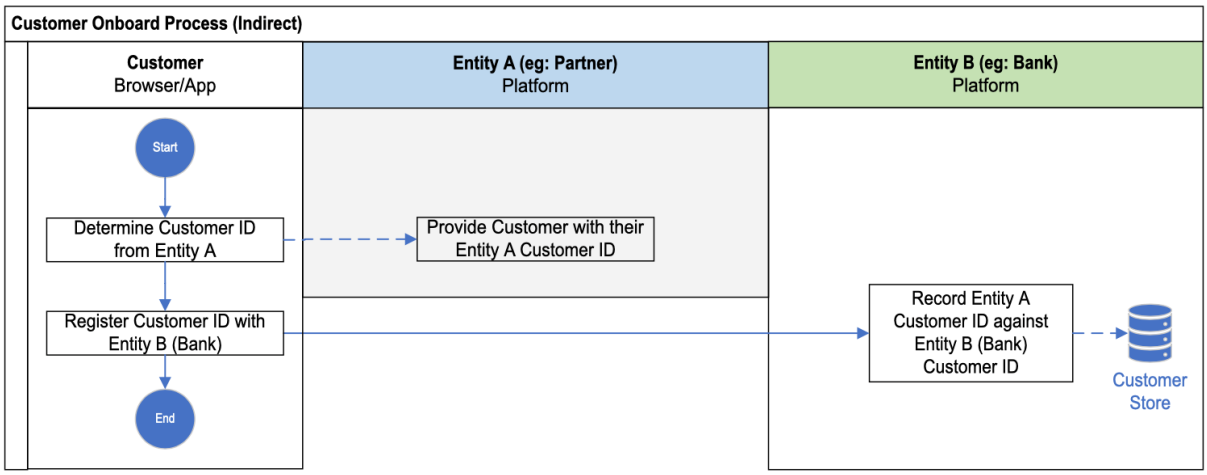
Indirect Onboarding

Assume that a Customer of Entity CB (Bank) wishes to consume Entity B's (Bank) services via Entity A. This Customer is already onboarded directly with both Entity A and Entity B (Bank). Therefore the Indirect Onboarding refers to the registration with Entity B (Bank) of the relationship between Customer and Entity A. This then allows Entity A to act on-behalf-of the Customer to Entity B (Bank).

Customer Onboarding

In order for Entity A to act on-behalf of (OBO) a Customer, Entity B must map Entity A's Customer ID to Entity B's (Bank) internal Customer ID.

Process Flow (Customer Onboarding)



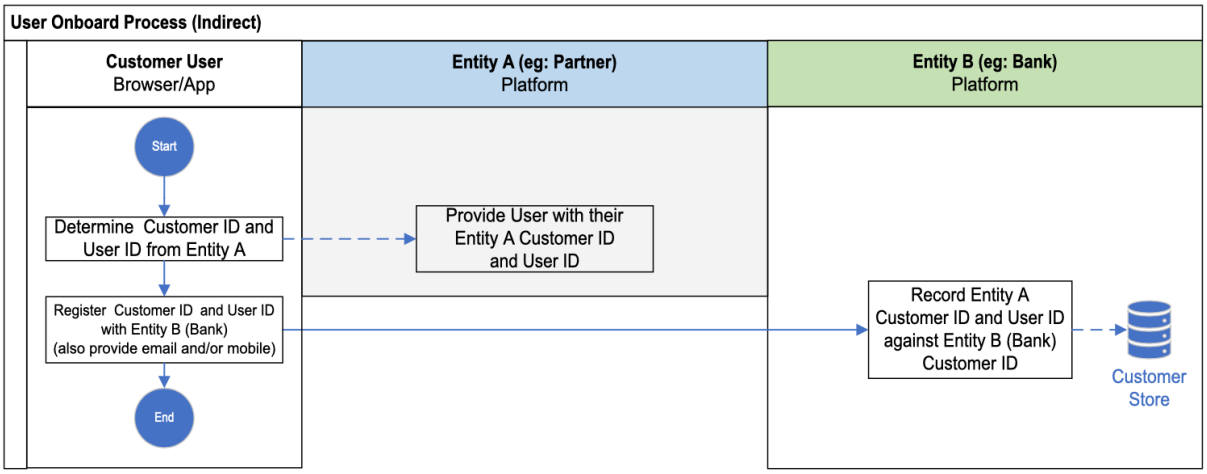
Process Steps (Customer Onboarding)

1. Customer retrieves their Customer ID from Entity A that Entity A uses to uniquely identify the Customer.
2. Customer registers their relationship with Entity B (Bank) using their Customer ID from Entity A.

User Onboarding

When certain transactions require multi-factor authentication, the onboarding of the Customer User is required.

Process Flow (User Onboarding)



Process Steps (User Onboarding)

1. Customer User retrieves their Customer ID and User ID from Entity A
2. Customer registers themselves with Entity B (Bank) using their Customer ID and User ID from Entity A, along with other user details such as email, mobile, etc..

Authentication

Authentication refers to two entities establishing trust to one another to facilitate the Trade Finance API traffic between them.

Unlike SWIFT, a closed community, where each entity being issued a unique identifier BIC (and MT798 being the messaging spec), Trade Finance API spec will be used in an open community, without a unique business entity identifier.

There are two (2) options at which Authentication can occur:

1. Mutual Transport Layer Security (mTLS)
2. Bearer Token (JWT)

It is recommended to use mTLS as it is the most efficient way to authenticate a client, given that it is embedded into the transport layer. It also means that other forms of transport (other than HTTPS) will still use the same mTLS handshake at connection time. Furthermore: HTTP Keep-Alive, or WebSocket (or any other form of multi-message TLS socket protocol) only requires one authentication process at connection; subsequent messages can be trusted.

However if an Entity cannot support mTLS, then the fallback is to the Bearer Token (JWT) as this can be used for both AuthN and AuthZ.

Mutual Transport Layer Security

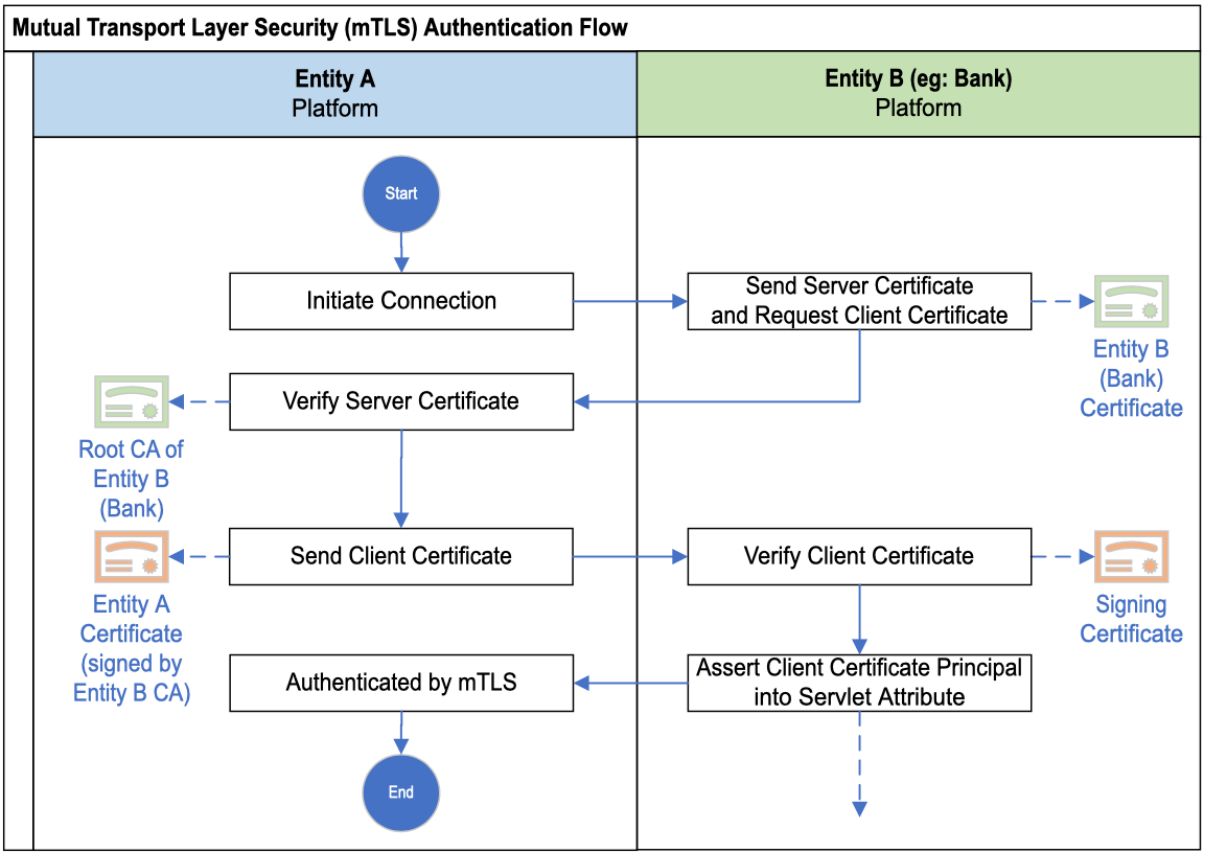
This is the standard Mutual TLS approach as described in the OAuth 2.0 standards (RFC 8705). mTLS makes use of industry standard X.509 Certificates for establishing trust and a secure (encrypted) channel between two Entities.

Regular TLS is widely used by websites that have the HTTPS protocol, however there is only a Server Certificate in this scenario. In other words, only Entity A can trust Entity B; Entity B has no way of identifying Entity A.

As part of the mTLS socket connection between client and server, both sides must present a valid Certificate that the counter-party can verify the certificate came from a trusted Certificate Authority.

Unlike key pairs, where public keys need to be exchanged between both parties to aid in mutual identity, certificates don't require any repository on the recipient (Entity B) side, because the certificate can be verified against the signing certificate. It also makes maintenance very straightforward as an expired certificate can be swapped for the new one at any time by the client without any synchronised maintenance on both sides.

Process Flow (mTLS)



Process Steps (mTLS)

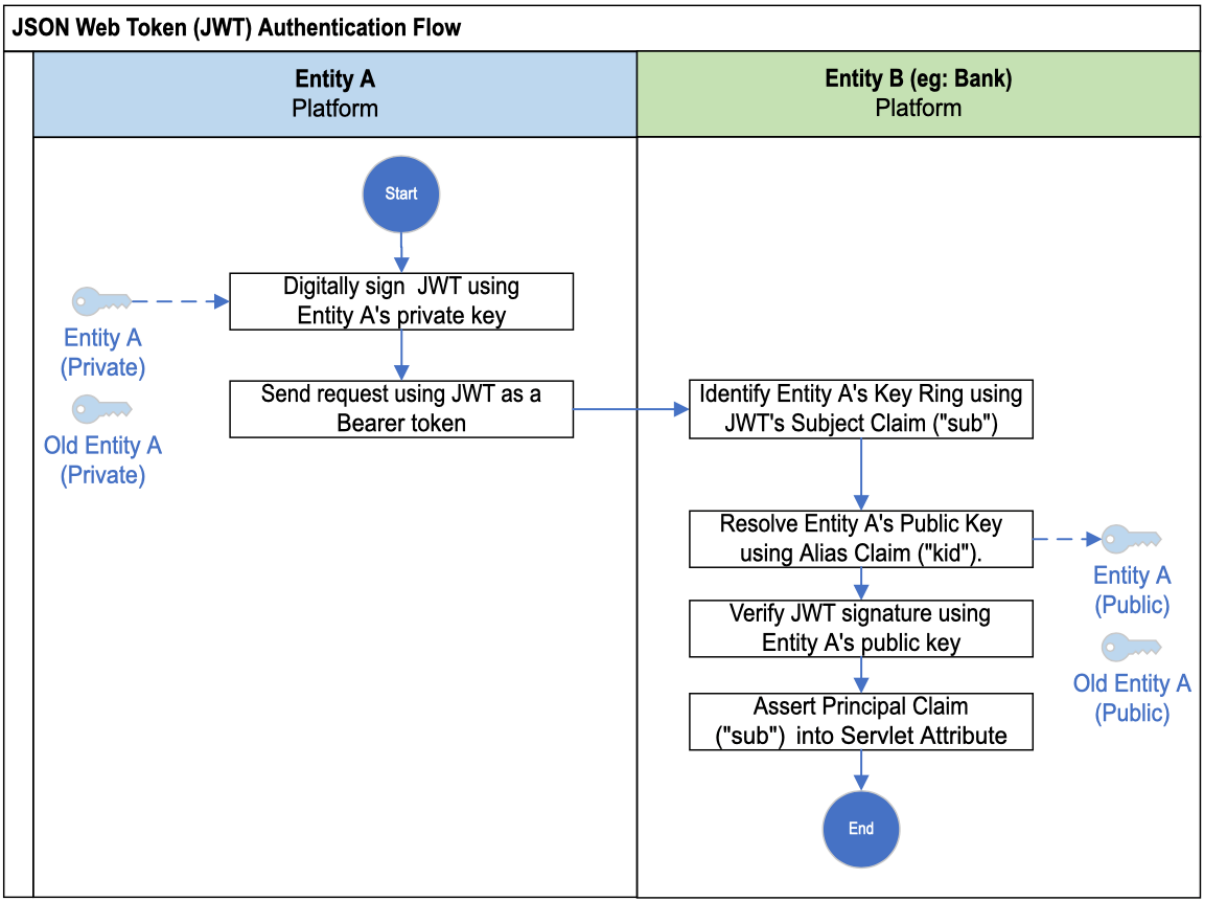
1. Entity A initiates a socket connection to Entity B (Bank)
2. Entity B requests the socket be mutually secure and sends their Server Certificate and requests the Client's Certificate
3. Entity A verifies Entity B's (Bank) certificate
4. Entity A sends their Client Certificate to Entity B (Bank)
5. Entity B verifies Entity A's certificate
6. Entity B (Bank) asserts the certificate into the Servlet Attribute so that its principal (Entity A's identity) can be extracted by an API service and entitlements enforced.
7. *(not shown) Symmetric Key is calculated on both sides in order to send encrypted traffic back and forth.*

Bearer Token

This is the standard Private Key JWT approach as described in the OAuth 2.0 standards (RFC 7521). A Bearer Token is typically a JSON Web Token (JWT) that is sent over a secure channel as part of the request's header. For an HTTPS request, this is the "Authorization" HTTP Header. A JWT is a digitally signed message using an RSA private key. The associated RSA public key can then be used to verify the digital signature to ensure the contents of the message can be trusted and has not been tampered with.

As part of the direct onboarding process, these RSA public keys are exchanged. Typically they are exchanged as part of a key ring, rather than the public key itself. A key ring is a set of public keys that are indexed by an alias. The alias can be specified in the Key Identifier ("kid") JWT header claim. That way the recipient can resolve which public key to use in the key ring. This is useful when transitioning between old and new keys.

Process Flow (JWT Bearer Token)



Process Steps (JWT Bearer Token)

1. Entity A builds a JWT and specifies the alias of the RSA key in the "kid" Header Claim. The JWT is digitally signed by the RSA private key.
2. Entity A sends a request to Entity B (Bank) with the JWT added as a Bearer Token in the message header. For HTTP, this is in the "Authentication" HTTP Header.
3. Entity B (Bank) will resolve the key ring for Entity A using the Subject Claim ("sub") in the JWT body.
4. **Entity B (Bank) will resolve the RSA Public key from Entity A's key ring using the Key Identifier Claim ("kid") in the JWT header.**
5. Entity B (Bank) will verify the digital signature of the JWT using the RSA Public Key.
6. Entity B (Bank) asserts the JWT claims into the Servlet Attribute so that identity and other preferences can be extracted by an API service and entitlements enforced.

Authorisation

This is the standard OAuth Assertions Framework approach as described in the OAuth 2.0 standards (RFC 7521). Authorization is implemented internally to an Entity as there is no use of a login service that issues a session token containing authorized details. Identity (Authentication) is all that's required to verify and secure communications between two Entities and it is up to the fulfilment of these requests to determine whether or not the client is authorised to do so.

However in the case of two-factor authentication where an Entity needs to verify that the transaction truly originated from a real Customer, then a token directly from the Customer such as a one-time-password (OTP) needs to accompany the request. The verification of the OTP therefore authorises the calling Entity to act on behalf of the Customer. To ensure the OTP is not tampered with, it is part of the Bearer token JWT. See the next section Identity Assertion for a deeper look into the JWT.

Identity Assertion

Identity Assertion comes in the form of an on-behalf-of ("obo") claim in a JSON Web Token (JWT), which is presented as a Bearer token with the request. Typically this will be part of the request header, which in the case of an HTTPS transport, the "Authorization" HTTP header. Other transports (such as a WebSocket, or an Event Bus) also have a similar header concept in the message and can all share the same Bearer token as a JWT concept. A JWT is digitally signed and verified by the RSA key pair (private and public key respectively) that is shared during onboarding.

The agreed claims in the JWT may differ from each Entity, however a sample set of claims may include:

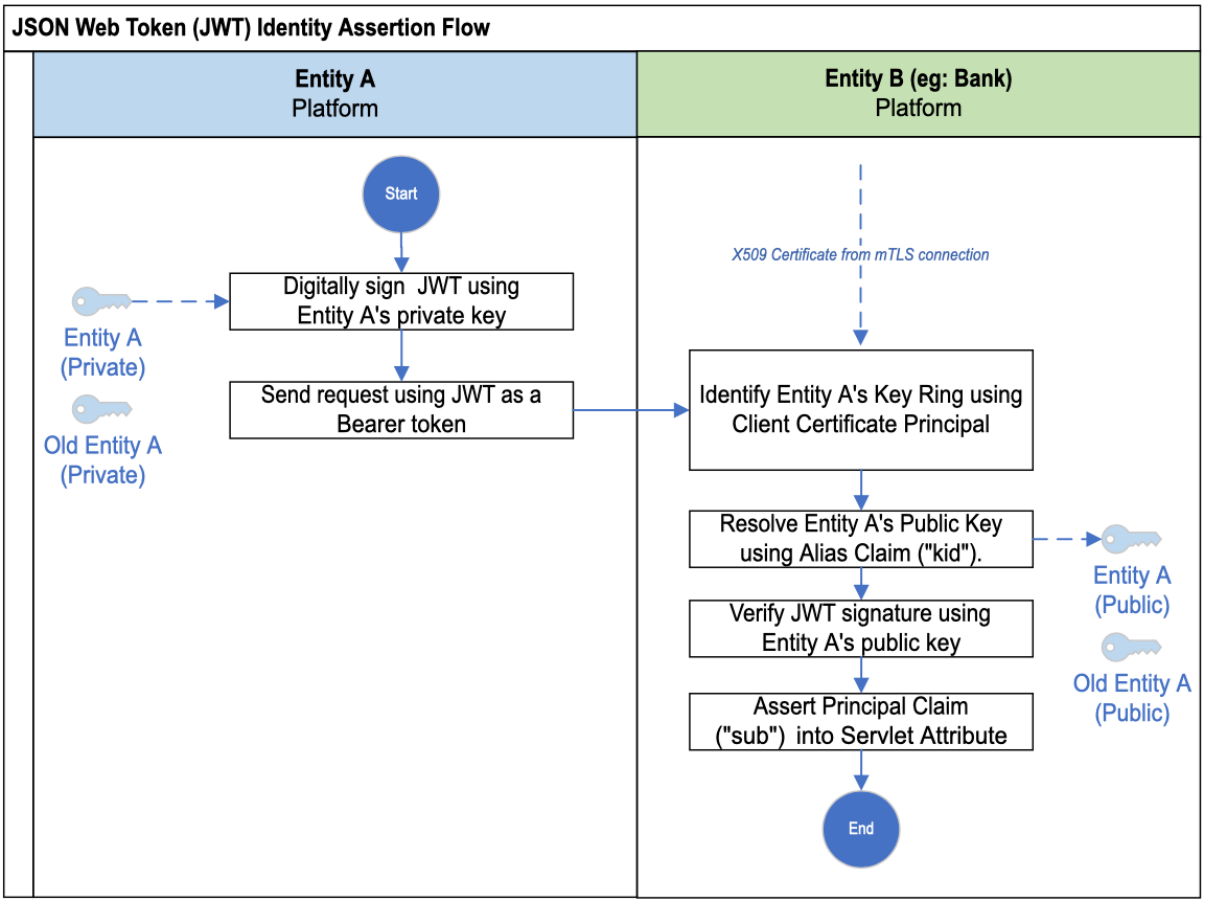
Claim	Claim Name	Example	Description
kid	Key Identifier	2022_key	Optional header claim. Alias of the public key when multiple may exist (for example during key swaps).
sub	Subject	P12345678	The Profile ID that determines the entitlements.
aud	Audience	ENTITY_B	The identifier of the system being called that can accept the request.
obo	On-Behalf-Of	CUST12345	Optional. The client's Customer ID.
uid	On-Behalf-Of User ID	user@customer.com	Optional. The client's User ID.

nbf	Not Before	16679600030	Epoc when the JWT becomes valid.
iat	Issued At	16679600030	Epoc when the JWT was generated.
exp	Expiration Time	16679600090	* Epoc when the JWT will expire.
jti	JWT ID	3159a01e-4c56-4b58-97c2-58f2021631ca	A client-generated unique identifier for the request. Supplied in the response.
sk	Secret Key	*<base64 string>*	Required if payload is encrypted. RSA encrypted secret key encoded into Base64.
iv	IV	*<base64 string>*	Required if payload is encrypted. Initial Vector for the encryption.
otp	One-Time Password	123456	Required if the API requires two-factor authentication.

** Entities will ensure the duration of a JWT is not longer than a short period of time (say one minute). Therefore the difference in time between "exp" and "nbf" (or "iat" if "nbf" is not specified) does not exceed 60 seconds.*

As for the actual entitlements that an Entity will grant another, these can remain internal.

Process Flow (Identity Assertion)



Process Steps (Identity Assertion)

1. Entity A builds a JWT and specifies the alias of the RSA key in the "kid" Header Claim. The JWT is digitally signed by the RSA private key.
2. Entity A sends a request to Entity B (Bank) with the JWT added as a Bearer Token in the message header. For HTTP, this is in the "Authentication" HTTP Header.
3. Entity B (Bank) will resolve the key ring for Entity A using the Subject Claim ("sub") in the JWT body.
4. **Entity B (Bank) will resolve the RSA Public key from Entity A's key ring using the Principal from the X.509 certificate used to initiate the mTLS connection.**
5. Entity B (Bank) will verify the digital signature of the JWT using the RSA Public Key.
6. Entity B (Bank) asserts the JWT claims into the Servlet Attribute so that identity and other preferences can be extracted by an API service and entitlements enforced.

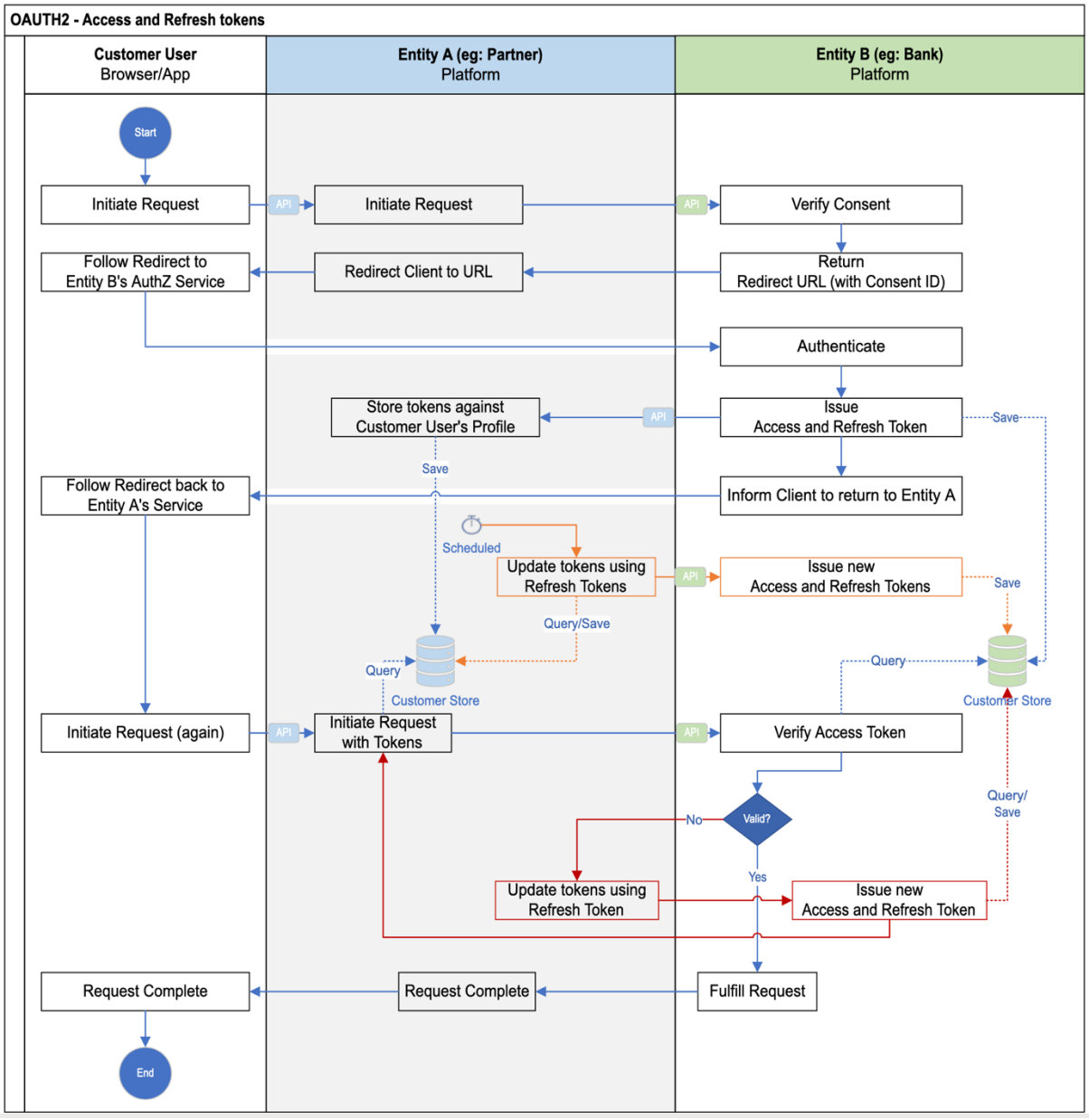
Customer Consent

When an Entity is acting on-behalf-of the Customer (B2B2B), there may be certain transactions that require the explicit consent of the Customer. This can be achieved using the Authorization Framework as described in the OAuth 2.0 standards (RFC 6749). There are two options available: With Session/Refresh Tokens or No Tokens.

Session/Refresh Tokens

Session/Refresh tokens are what Entity A can use to act on-behalf-of a Customer. They are passed directly from Entity B (Bank) to Entity A upon authentication of the Customer, as an added layer of security. The Session token is used in each API call, whereas the Refresh token is used to obtain a new Session token that has (or is close to) expiring. Session tokens are typically short-lived, whereas a Refresh token is long-lived. The Customer or Entity B (Bank) can at any time can revoke access by deleting/expiring the tokens in Entity B (Bank).

Process Flow (Session/Refresh Tokens)



Process Steps (Session/Refresh Tokens)

1. Customer User initiates a request via Entity A's UI.
2. Entity A initiates a request to Entity B (Bank) on-behalf-of Customer User.
3. Entity B (Bank) determines no consent (Access Token) exists for Customer User, so replies with a new Consent ID associated with the request and a Redirect URL for the Customer User to visit.
4. Entity A replies to the Customer User, instructing a redirect to Entity B (Bank)'s Redirect URL with the Consent ID as a parameter.
5. Customer User follows the Redirect URL + Consent ID to Entity B's (Bank) portal (with the return URL to Entity A as a parameter).

6. Entity B (Bank) authenticates Customer User (eg: Login, 2FA, etc).
7. Entity B (Bank) generates and records a set of Access and Refresh tokens for Entity A to act on-behalf-of the Customer User.
8. Entity B (Bank) calls Entity A's API, informing them of the consent given with the tokens included.
9. Entity A stores the Access and Refresh tokens against the Customer User's profile.
10. Entity B (Bank) replies to Customer User, providing a Redirect URL back to Entity A's service.
11. Customer User retries the request that was done in Step 1.
12. Entity A retries the request to Entity B (Bank), this time with the Access token.
13. Entity B (Bank) verifies the Access token is valid
    a. Yes, is valid:
        i. Entity B (Bank) fulfils the request and returns the result back to Entity A, which in turn returns a result back to the Customer User.
    b. No, is invalid (steps shown in red):
        i. Entity A requests Entity B (Bank) to refresh the tokens using the Refresh token.
        ii. Entity B (Bank) verifies the Refresh token and issues a new Access and Refresh Token. If Refresh token is invalid, Entity A is denied access.
        iii. Entity B (Bank) returns new Access and Refresh tokens.
        iv. Entity A repeats the initiate request with the Access token (step 12).

Process Steps (Entity A's Token Refresh Service)

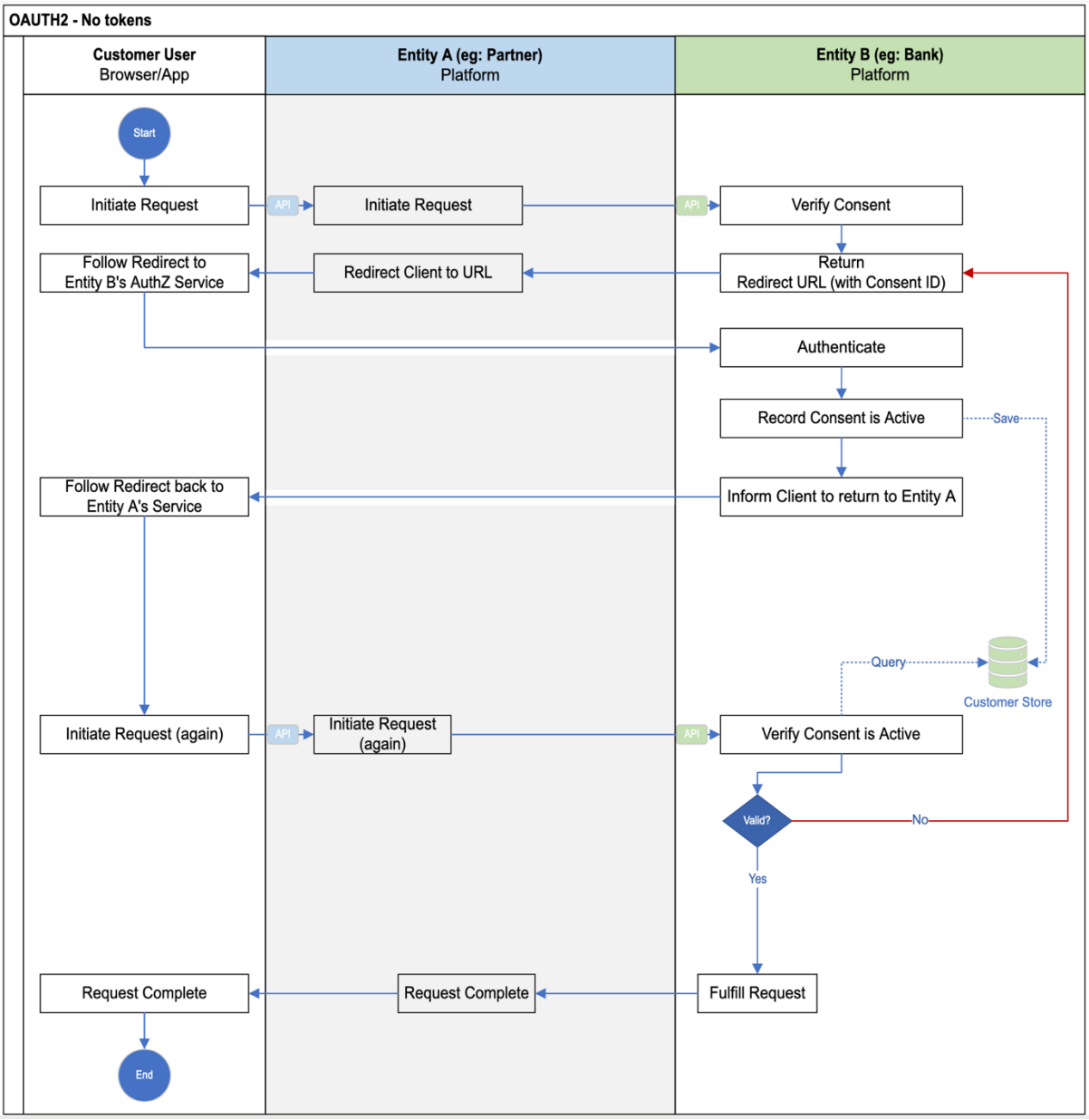
Optional scheduled process to keep all access tokens valid (steps shown in orange).

1. Entity A cycles through all stored Refresh tokens for active Customer Users and for-each:
    a. Entity A calls Entity B's (Bank) token refresh API.
    b. Entity B (Bank) verifies Refresh token and issues a new set of Access and Refresh tokens.

No Tokens

No Tokens follows the same Session/Refresh token approach, but it is simplified in that no tokens are needed. Therefore Entity A requires no API endpoint to receive tokens from Entity B (Bank) and Entity A requires no storage or token refresh capability. The consent is remembered by Entity B (Bank) and the Customer User or Entity B (Bank) can still at any time can revoke access by deleting/expiring the consent record in Entity B (Bank).

Process Flow (No Tokens)



Process Steps (No Tokens)

1. Customer User initiates a request via Entity A's UI.
2. Entity A initiates a request to Entity B (Bank) on-behalf-of Customer User.
3. Entity B (Bank) determines no consent record exists for Customer User, so replies with a new Consent ID for that Customer User's request and a Redirect URL for the Customer User to visit.
4. Entity A replies to the Customer User, instructing a redirect to Entity B (Bank)'s Redirect URL + Consent ID, with a return URL to Entity A as a parameter.
5. Customer User follows the Redirect URL + Consent ID to Entity B's (Bank) portal.
6. Entity B (Bank) authenticates Customer User (eg: Login, 2FA, etc).
7. Entity B (Bank) stores the consent for Entity A to act on-behalf-of Customer User for a pre-determined time.
8. Entity B (Bank) replies to Customer User, providing a Redirect URL back to Entity A's service.
9. Customer User retries the request that was done in step 1.
10. Entity A retires the request that was done in step 2.
11. Entity B (Bank) verifies the consent is active
    a. Yes, is valid:
        i. Entity B (Bank) fulfils the request and returns the result back to Entity A, which in turn returns a result back to the Customer User.
    b. No, is invalid:
        i. Entity B (Bank) repeats the request to redirect the Customer User (step 3)

API Request/Response

TLS-only

A TLS-only request uses the JWT verify process as the AuthN. Key differences between TLS and mTLS are highlighted in ***bold*** *in the process steps.*

For a sender (Entity A) an API call requires the use of the following components:

1. Secret Key
   Uniquely generated using the AES algorithm that is used for encrypting and decrypting the request and response payloads respectively.

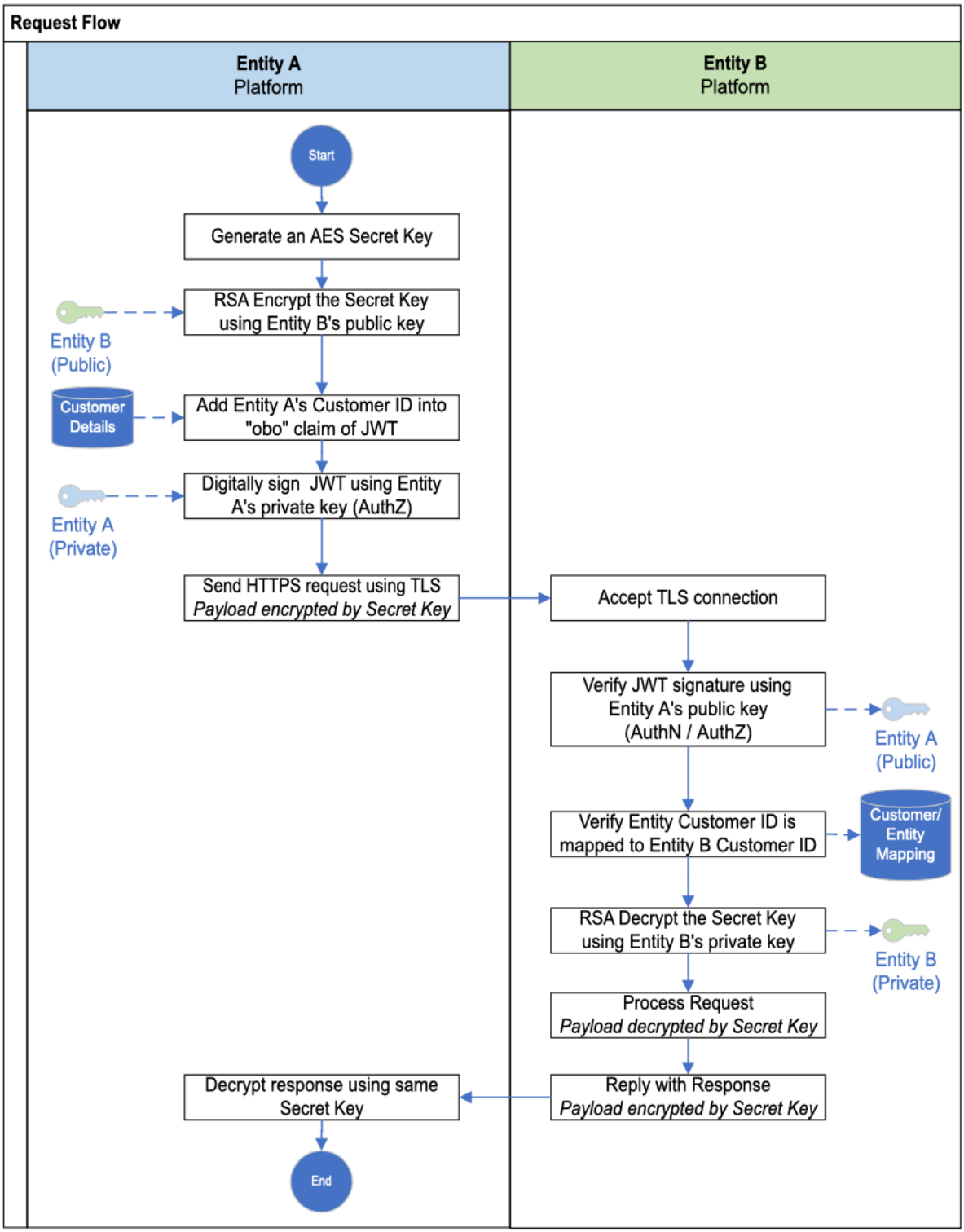
2. RSA Private Key
   Used to sign the JWT that contains details we want to ensure isn't tampered with.
3. Recipient's RSA Public Key
   Used to encrypt the Secret Key above.

For a recipient (Entity B) an API call requires the use of the following components:

1. Sender's RSA Public Key
   Used to verify the JWT digital signature, so that the contents can be trusted.
2. RSA Private Key
   Used to decrypt the Secret Key sent in the header (or JWT) of the request.

For details on the encryption/decryption of the payload, see Payload Encryption below.

Process Flow (API over TLS)



Process Steps

1. Entity A generates an AES Secret Key (key size of 256).
2. Entity A RSA encrypts the Secret Key using the recipient's (Entity B) Public Key.

3. Entity A generates a 12-byte random Initial Vector (IV).
4. Entity A prepares the encrypted Secret Key and IV as a request header (or JWT claim).
5. Entity A builds the JWT and includes Entity A's Customer ID into the on-behalf-of ("obo") claim of the JWT.
6. Entity A digitally signs the JWT using their own Private Key.
7. Entity A encrypts the payload using the "AES/GCM/NoPadding" transformation, Secret Key and the IV.
8. **Entity A initiates a TLS connection to Entity B.**
9. **Entity B accepts the TLS connection from Entity A.**
10. **Entity B verifies the digital signature on the JWT using Entity A's Public Key (ie- Authentication).**
11. Entity B determines Entity B's Customer ID that the transaction is on behalf of by looking up the mapping of Entity A's Customer ID to Entity B's Customer ID, where Entity A's Customer ID is in the on-behalf-of ("obo") claim in the JWT.
12. Entity B RSA decrypts the Secret Key using their own Private Key.
13. Entity B processes the request and decrypts the payload using the AES Secret Key.
14. Entity B encrypts the response using the same AES Secret Key.
15. Entity A decrypts the response using the same AES Secret Key.

Mutual Authentication (mTLS)

*An mTLS request uses the client certificate as the AuthN. Key differences between TLS and mTLS are highlighted in **bold** in the process steps.*

For a sender (Entity A) an API call requires the use of the following components:

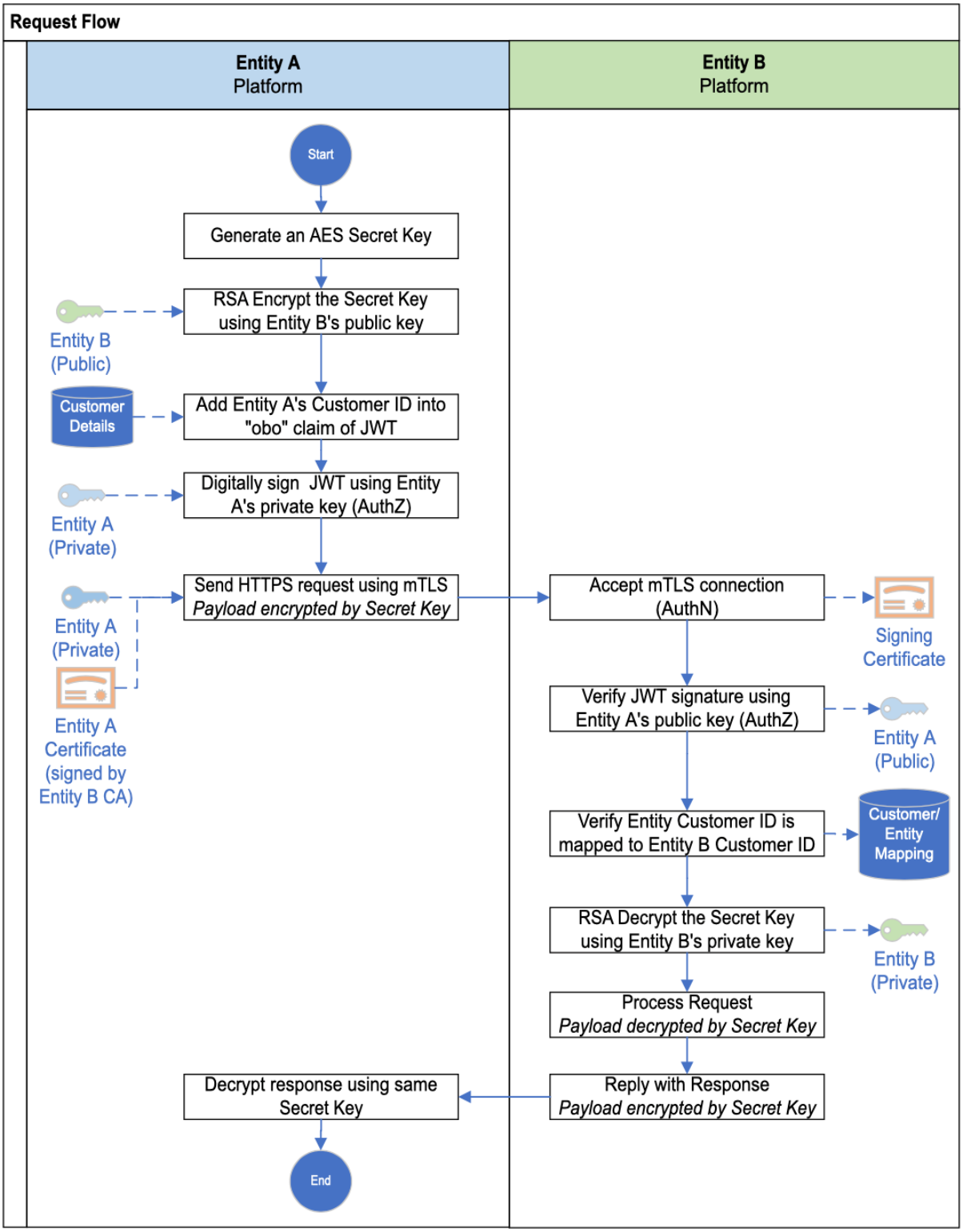
1. Certificate
   Created by the recipient's certificate authority to initiate a mutual TLS connection between both Entities, along with the original RSA private key used to generate the original certificate request.
2. Secret Key
   Uniquely generated using the AES algorithm that is used for encrypting and decrypting the request and response payloads respectively.
3. RSA Private Key
   Used to sign the JWT that contains details we want to ensure isn't tampered with.
4. Recipient's RSA Public Key
   Used to encrypt the Secret Key above.

For a recipient (Entity B) an API call requires the use of the following components:

1. Signing Certificate
   Root or intermediate certificate used to generate certificates from certificate requests.
2. Sender's RSA Public Key
   Used to verify the JWT digital signature, so that the contents can be trusted.
3. RSA Private Key
   Used to decrypt the Secret Key sent in the header (or JWT) of the request.

For details on the encryption/decryption of the payload, see Payload Encryption below.

Process Flow (API over mTLS)



Process Steps (API over mTLS)

1. Entity A generates an AES Secret Key (key size of 256).

2. Entity A RSA encrypts the Secret Key using the recipient's (Entity B) Public Key.
3. Entity A generates a 12-byte random Initial Vector (IV).
4. Entity A prepares the encrypted Secret Key and IV as a request header (or JWT claim).
5. Entity A builds the JWT and includes Entity A's Customer ID into the on-behalf-of ("obo") claim of the JWT.
6. Entity A digitally signs the JWT using their own Private Key.
7. Entity A encrypts the payload using the "AES/GCM/NoPadding" transformation, Secret Key and the IV.
8. **Entity A initiates an mTLS connection to Entity B using the certificate signed by Entity B's certificate authority.**
9. **Entity B verifies the mTLS connection by ensuring Entity A's certificate came from their own Signing Certificate (ie- Authentication).**
10. Entity B verifies the digital signature on the JWT using Entity A's Public Key.
11. Entity B determines Entity B's Customer ID that the transaction is on behalf of by looking up the mapping of Entity A's Customer ID to Entity B's Customer ID, where Entity A's Customer ID is in the on-behalf-of ("obo") claim in the JWT.
12. Entity B RSA decrypts the Secret Key using their own Private Key.
13. Entity B processes the request and decrypts the payload using the AES Secret Key.
14. Entity B encrypts the response using the same AES Secret Key.
15. Entity A decrypts the response using the same AES Secret Key.

Security Requirements

1. All communication must be over a secure socket TLS v1.2 (or higher).
2. It is preferred to authenticate using mTLS (client-side certificate), however a Bearer token (JWT) that is digitally signed by a private key is an acceptable alternative.
3. Message bodies containing sensitive information should be encrypted using an AES Secret Key.
4. Message headers containing sensitive information should be encrypted using RSA (eg: the secret key used to encrypt the message body).
5. The AES Secret Key must be unique for each encrypted request and the synchronous encrypted response can use the same AES Secret Key, however asynchronous encrypted responses must use their own unique AES Secret Key.

6. All private keys must be stored outside of the application in a secure vault (eg: HSM).
7. Certificates and public keys are not required to be contained in a secure vault.
8. Key Pairs and Certificates must not have a validity of more than one (1) year.
9. Intermediate and root signing certificates are safe to have long term validity.
10. When a message payload is a document (or contains a document), it must be virus scanned as part of the inbound refinery process of the content repository or just before the document is persisted for further use. When a document is a multi-file archive (eg: ZIP), each document inside must be scanned individually.

Appendix

Sample Code

The following is a simple Java program that does the following:

1. Loads server public key from “server_test_key_public.pem” file.
2. Loads and parses client private key from “client_test_key_pcks8.pem” file.
3. Generates a Secret Key using the AES algorithm.
4. Encrypts the Secret Key using the RSA algorithm and server’s public key.
5. Generates a random 12-byte Initialisation Vector for GCM cipher.
6. Loads the payload from the “payload.json” file.
7. Encrypts the payload using the "AES/GCM/NoPadding" transformation, Secret Key and IV.
8. Generate a JWT that expires in one-minute that contains the Encrypted Secret Key, IV, the algorithms/transformations used, the asserted identity, on-behalf-of details, etc.
9. Digitally sign the JWT using the client private key and the RSA 256 algorithm.
10. POST a HTTP request to “https://localhost:8080/context-root/some-api".
11. If the response code is OK (200), then decrypt the response using the same Secret Key and IVand save the payload to “response.json”.

Test.java

```
package com.test;

import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.security.Key;
import java.security.KeyFactory;
import java.security.SecureRandom;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.time.Duration;
import java.time.Instant;
import java.time.temporal.ChronoUnit;
import java.util.Base64;
import java.util.Date;
import java.util.Locale;
import java.util.UUID;
import javax.crypto.Cipher;
import javax.crypto.CipherInputStream;
import javax.crypto.CipherOutputStream;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.GCMParameterSpec;

public class Test {

  private static final String PAYLOAD_TRANSFORMATION = "AES/GCM/NoPadding";
  private static final String SECRET_KEY_ALGORITHM = "AES";
  private static final String SECRET_KEY_TRANSFORMATION = "RSA";
  private static final SignatureAlgorithm JWT_SIGNING_ALGORITHM = SignatureAlgorithm.RS256;

  public static void main(String[] args) throws Exception {

    // Load recipient public key from a file
    X509EncodedKeySpec recipientPublicKeySpec = new X509EncodedKeySpec(
     Files.readAllBytes(Paths.get("server_test_key_public.pem"))
    );
    KeyFactory recipientKeyFactory = KeyFactory.getInstance("RSA");
    Key recipientPublicKey = recipientKeyFactory.generatePublic(recipientPublicKeySpec);

    // Load client private key from a file
    StringBuilder privateKeyContent = new StringBuilder();
    for (String line : Files.readAllLines(Paths.get("client_test_key_pcks8.pem")))
      if ((line.length() > 0) && (!(line.startsWith("-----"))))
        privateKeyContent.append(line);
    PKCS8EncodedKeySpec myPrivateKeySpec = new PKCS8EncodedKeySpec(
     Base64.getDecoder().decode(privateKeyContent.toString())
    );
    KeyFactory myKeyFactory = KeyFactory.getInstance(SECRET_KEY_TRANSFORMATION);
    Key myPrivateKey = myKeyFactory.generatePrivate(myPrivateKeySpec);
```

```
// Generate a secret key to be used to encrypt the payload
KeyGenerator keyGenerator = KeyGenerator.getInstance(SECRET_KEY_ALGORITHM);
keyGenerator.init(256);
SecretKey secretKey = keyGenerator.generateKey();

// Encrypt the secret key using the recipient public key
Cipher secretKeyEncryptionCipher = Cipher.getInstance(SECRET_KEY_TRANSFORMATION);
secretKeyEncryptionCipher.init(Cipher.ENCRYPT_MODE, recipientPublicKey);
String encryptedSecretKey = Base64.getEncoder().encodeToString(
  secretKeyEncryptionCipher.doFinal(secretKey.getEncoded())
);

// Generate a random Initial Vector
byte[] iv = new byte[12];
(new SecureRandom()).nextBytes(iv);
String ivString = Base64.getEncoder().encodeToString(iv);

// The payload raw input stream
InputStream payloadIn = Files.newInputStream(Paths.get("payload.json"));

// Wrap the request input stream with an encryption stream
Cipher payloadEncryptionCipher = Cipher.getInstance(PAYLOAD_TRANSFORMATION);
payloadEncryptionCipher.init(Cipher.ENCRYPT_MODE, secretKey, new GCMParameterSpec(
  128, iv)
);
CipherInputStream encryptedIn = new CipherInputStream(
  payloadIn, payloadEncryptionCipher
);

// Unique identify for the transaction.
String myClientId = UUID.randomUUID().toString();

// Issued time (with expiry being one-minute after)
Instant now = Instant.now();

// Create the JWT
String jwtTokenString = Jwts.builder()
    .setSubject("P0000123456")
    .setAudience("ENTITY_B")
    .setId(myClientId)
    .setIssuedAt(Date.from(now))
    .setExpiration(Date.from(now.plus(1L, ChronoUnit.MINUTES)))
    .claim("obo", "CUST_1234")
    .claim("uid", "CUST_USER_456")
    .claim("otp", "123456")  // one-time password
    .claim("iv", ivString)
    .claim("tf", PAYLOAD_TRANSFORMATION)
    .claim("sk", encryptedSecretKey)
    .claim("ska", SECRET_KEY_ALGORITHM)
    .claim("skt", SECRET_KEY_TRANSFORMATION)
    .claim("loc", Locale.getDefault().toString())
    .claim("ver", "1")
    .signWith(myPrivateKey, JWT_SIGNING_ALGORITHM)
    .setHeaderParam("kid", "client_test_key_public")
    //.compressWith(CompressionCodecs.GZIP)
    .compact()
;
```

```
    System.out.println("JWT=" + jwtTokenString);

    // Submit the request
    HttpClient httpClient = HttpClient.newHttpClient();
    HttpRequest httpRequest = HttpRequest.newBuilder(
      URI.create("https://localhost:8080/context-root/some-api"))
        .POST(HttpRequest.BodyPublishers.ofInputStream(() -> encryptedIn))
        .timeout(Duration.ofSeconds(60))
        .header("Authorization", "Bearer " + jwtTokenString)
        .build()
    ;
    encryptedIn.close();
    payloadIn.close();

    // Process the response
    HttpResponse<InputStream> httpResponse = httpClient.send(
      httpRequest, HttpResponse.BodyHandlers.ofInputStream()
    );

    // Stream the response to a file if successful
    if (httpResponse.statusCode() == 200) {

      // The response raw output stream
      OutputStream responseOut = Files.newOutputStream(Paths.get("response.json"));

      // Wrap the response output stream with a decryption stream
      Cipher responseDecryptionCipher = Cipher.getInstance(PAYLOAD_TRANSFORMATION);
      responseDecryptionCipher.init(
        Cipher.DECRYPT_MODE, secretKey, new GCMParameterSpec(128, iv)
      );
      CipherOutputStream decryptedOut = new CipherOutputStream(
        responseOut, responseDecryptionCipher
      );

      // Write the file
      httpResponse.body().transferTo(decryptedOut);
      decryptedOut.close();
      responseOut.close();
    }
  }
}
```

Key Pair Generation

The following is a walkthrough of how to generate a key pair using “openssl”.

1. Generate a private key

   openssl genrsa -des3 -out private.pem 2048

2. Export a public key from the private key

   openssl rsa -in private.pem -outform PEM -pubout -out public.pem

mTLS (server-side)

The following is a walkthrough of how to setup a Java SpringBoot application to enforce mTLS. It makes use of "openssl" to generate and sign keys/certificates.

1. Generate a Private Key and Root Certificate Authority that can be used for signing both the server and client certificates.

```
openssl req -x509 -sha256 -days 3650 -newkey rsa:4096 \
 -keyout rootCA.key -out rootCA.crt
```

2. Generate a Private Key and Certificate Request for the server certificate generation that will run on "localhost".

```
openssl req -new -newkey rsa:4096 -keyout localhost.key \
 –out localhost.csr
```

3. Create a configuration file called "localhost.ext" for signing the "localhost" certificate that contains the following:

```
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
subjectAltName = @alt_names
[alt_names]
DNS.1 = localhost
```

4. Generate the certificate for "localhost" from the Certificate Request and using the Root Certificate Authority

```
openssl x509 -req -CA rootCA.crt -CAkey rootCA.key \
 -in localhost.csr -out localhost.crt -days 365 -CAcreateserial \
 -extfile localhost.ext
```

5. Package the "localhost" certificate and private key into a PKCS12 archive

```
openssl pkcs12 -export -out localhost.p12 -name "localhost" \
 -inkey localhost.key -in localhost.crt
```

6. Import the "localhost" package into a JKS keystore, which allows the server to have TLS.

```
keytool -importkeystore -srckeystore localhost.p12 \
```

```
-srcstoretype PKCS12 -destkeystore keystore.jks -deststoretype JKS
```

7. Import the Root Certificate Authority certificate into a JKS truststore, which allows the server to have mTLS.

```
keytool -import -trustcacerts -noprompt -alias ca \
 -ext san=dns:localhost,ip:127.0.0.1 -file rootCA.crt \
 -keystore truststore.jks
```

8. Configure the Java SpringBoot service to use the "localhost" certificate for TLS and the truststore for enforcing mTLS. "server.ssl.client-auth=need" will enforce mTLS, where as "server.ssl.client-auth=want" will make mTLS optional.

```
server.ssl.key-store=keystore.jks
server.ssl.key-store-password=changeit
server.ssl.key-alias=localhost
server.ssl.key-password=changeit
server.ssl.enabled=true
server.ssl.trust-store=truststore.jks
server.ssl.trust-store-password=changeit
server.ssl.client-auth=need
```

mTLS (client-side)

When using a standard Key Store and Trust Store for a Java client, an mTLS connection will cause the Key Manager to pick the first certificate it finds in the Key Store that was signed by the specified issuer. If there happens to be multiple certificates issued, then choosing the right one is not possible. Also a client may want to store certificates somewhere other than a regular Key Store, as any changes to the Key Store would require a restart of the service, which may not be practical.

The standard approach is to initialise an SSL Context that has a customised Key Manager and Trust Manager, so that the certificate resolver can be overridden. The code snippet below shows how to include a custom Key Manager and Trust Manager into the SSL Context used for an HTTP request:

```
// Setup a customised SSL Context
SSLContext sslContext = SSLContext.getInstsance("TLS");
sslContext.init(
  new KeyManager[] { myKeyManager }
  , new TrustManager[] { myTrustManager }
  , SecureRandom.getInstanceStrong()
);
```

```
// Create a HTTP Client using the custom SSL Context
HttpClient httpClient = HttpClient.newBuilder().sslContext(sslContext).build();

// Submit the request
HttpRequest httpRequest = HttpRequest.newBuilder(
  URI.create("https://localhost:8080/context-root/some-api"))
    .POST(HttpRequest.BodyPublishers.ofInputStream(() -> encryptedIn))
    .timeout(Duration.ofSeconds(60))
    .header("Authorization", "Bearer " + jwtTokenString)
    .build()
;
```

About the International Chamber of Commerce

The International Chamber of Commerce (ICC) is the institutional representative of more than 45 million companies in over 100 countries. ICC's core mission is to make business work for everyone, every day, everywhere. Through a unique mix of advocacy, solutions and standard setting, we promote international trade, responsible business conduct and a global approach to regulation, in addition to providing market-leading dispute resolution services. Our members include many of the world's leading companies, SMEs, business associations and local chambers of commerce.