API Specification & Technical Integration Guide of HSBC QR Code Payment in Thailand

Version: 1.0

Purpose of this document

This document provide the audience with **OpenAPI specification** for describing REST APIs of HSBC Collection of digital payments - HSBC Omni Collect in the UK.

The target audience of this document is the Developer, Business Analyst and other related Project Team Member (who has the basic technical know-how of Web technology such as REST or JSON) of HSBC's client (i.e. the Merchant)

Update Log

- [Aug 4, 2021] **v1.0** Initial Draft

How to Read this Document

First of all, get to know what features we can offer in Features Overview and then get the key idea of our API operations in here. Before you connect to our API, remember to collect all prerequisites and go through all requirements in How to Connect section. Let's get started!

Features Overview

Webhooks for Payment

Asynchronous callback will be pushed back to Merchant regarding to different payment events such as a captured payment or a settled payment. Please see Webhooks for details.

How to Connect

API Connectivity refers to all measures and their components that establishes connection between HSBC, the API Provider and Merchant, the API Consumer.

	Definition	Components
API Authentication	HTTP BASIC Authentication	• Username • Password
	Locate API Gateway Policy of the corresponding user	• Client ID • Client Secret
User Identification	A Merchant Profile	• Merchant ID • Merchant Profile
Connection Security	HTTPS Connection (TLS 1.2) and Network Whitelisting	• SSL Certificate • Network Whitelist
Message Security	Digital Signing and Data Encryption	• A pair of Private Key & Public Key Certificate (PKI Model) • JWS Key ID • JWE Key ID

API Gateway URL

You need to include this before each API endpoint to make API calls.

Production
https://cmb-api.hsbc.com.hk/glcm-mobilecoll-mcth-ea-merchantservices-prod-proxy/v1

Sandbox
https://devclustercmb.api.p2g.netd2.hsbc.com.hk/glcm-mobilecoll-mcth-ea-merchantservices-cert-proxy/v1

API Authentication

Username & Password		
Purpose	All APIs are authorized using `Basic Authorization`	
Components	• Username	• Password
Where to get it?	Delivered by HSBC via secure email during onboarding procedure	
Implementation	In HTTP header: `Authorization: Basic [Base64-encoded Credential]`	

Client ID & Client Secret		
Purpose	API Gateway locates the corresponding policy of the specific API consumer	
Components	• Client ID	• Client Secret
Where to get it?	Delivered by HSBC via secure email during onboarding procedure	
Implementation	In HTTP header: `x-hsbc-client-id: [Client ID]`	In HTTP header: `x-hsbc-client-secret: [Client Secret]`

User Identification

Merchant Profile & Merchant ID		
Purpose	• Merchant Profile contains all necessary information from a Merchant in order to enable payment service.	• Merchant ID is used for Merchant identification in each API call.
Components	• Merchant Profile	• Merchant ID

Where to get it?	• Set up by HSBC team after collect information from Merchant	• Delivered by HSBC via secure email during onboarding procedure
Implementation	*nil*	In HTTP header: `x-hsbc-msg-encrypt-id: [Merchant ID]+[JWS ID]+[JWE ID]`

Connection Security

SSL Certificate & Network Whitelist			
Purpose	• Request HSBC API over HTTPS connection (TLS 1.2)	• Accept Callback API request over HTTPS connection (TLS 1.2)	
Components	• Public SSL Certificate issued by HSBC	• Merchant's web server or domain whose HTTPS connection is enabled	• Network Whitelist on HSBC system
Where to get it?	• Downloaded automatically by Browsers or API Tools, if any problem found, please contact HSBC	*nil*	*nil*
Implementation	*nil*	*nil*	• Merchant's domain URL will be configured in HSBC's network whitelist by HSBC team

Message Security - Data Encryption and Signing

On top of the Transport Layer Security, HSBC adopts additional security on the message being passed through the connection session. Data Encryption actually serves as a locked briefcase containing the data (the API message) within the HTTPS "tunnel". In other word, the communication has double protection.

> **DO YOU KNOW?**

> ! Javascript Object Signing and Encryption **(JOSE™)**, is a framework intended to provide methods to securely transfer information between parties. The JOSE framework provides a collection of specifications, including JSON Web Signature **(JWS™)** and JSON Web Encryption **(JWE™)**, to serve this purpose.

HSBC uses JWS to sign message payload and JWE to encrypt the signed message while these two objects are created by using a pair of Private Key & Public Key Certificate (PKI Model).

Private Key & Public Key Certificate (PKI Model)		
Purpose	• Digitally sign a API request message • Decrypt a API response message	• Encrypt the signed API request message • Verify a signed API response message
Components	• Private Key issued by Merchant	• Public Key Certificate issued by HSBC
Where to get it?	• Created by any Public Key Infrastructure (PKI) toolkits, such as Keytool™ and OpenSSL™. Technical detail is in here	• Exchanged with HSBC with the Public Key Certificate issued by Merchant
Implementation	Please see the technical detail in here	

> ! **NOTICE:**
> Technically, X.509 certificate can be served as a SSL Certificate as well as a Public Key Certificate for Data Encryption. However, HSBC recommends Merchant to use a different X.509 Certificate for Data Encryption for segregation of certificate usage.
>
> Moreover, the Public Key Certificate does not have to be CA-signed. However, if Merchant decides to enhance security, a CA-Signed Certificate is always welcome.

keyID of JWS™ & JWE™		
Purpose	• The unique identifier to bind Merchant's Private Key in order to create a JWS object - a signed Message Payload	• The unique identifier to bind HSBC's Public Key Certificate in order to create a JWE object - an encrypted JWS object
Components	• keyID of JWS™	• keyID of JWE™
Where to get it?	• Mutual agreed between Merchant and HSBC	• Mutual agreed between Merchant and HSBC
	• Define in program coding, see demo in here, and;	

Implementation	• In HTTP header: `x-hsbc-msg-encrypt-id: [Merchant ID]+[JWS ID]+[JWE ID]`

> ! **NOTICE:**
> For security purposes, `HSBC's Public Key Certificate` and its associated `keyID` will be renewed **every** year and a Certificate Renewal process will be triggered. More detail is covered in section Key Renewal

How to Sign and Encrypt Outgoing Message

Every message sent to HSBC must be signed and encrypted. From the point of view of a Merchant, an **Outgoing Message** means:

- the Request Message of a Normal API, or
- the Respond Message of a Callback API.

To help you understand how to construct a Signed and Encrypted Message, let's take the Java program below as an example. Do not worry if you are not familiar with Java, the idea is to let you know the steps and all needed components:

> ! **NOTICE:** These Java codes are for demonstration only and it's not *plug and play*.

```
private JWSObject signMessage(String messagePayload, KeyStore ks, String keyAlias, String keyPw)
  throws UnrecoverableKeyException, KeyStoreException, NoSuchAlgorithmException, JOSEException {
#1  Payload payload = new Payload(messagePayload);

#2  JWSHeader header = new JWSHeader
                .Builder(JWSAlgorithm.RS256)
                .keyID("0001")
                .customParam("iat", Instant.now().getEpochSecond()).build();
#3  JWSObject jwsObject = new JWSObject(header, payload);

#4  PrivateKey privateKey = (PrivateKey) ks.getKey(keyAlias, keyPw.toCharArray());
    JWSSigner signer = new RSASSASigner(privateKey);
#5  jwsObject.sign(signer);

    return jwsObject;
}
```

1. Prepare your **Message Payload**, that is, the plain `json` request message
2. Create **JWS Header** where the parameters are as follows:

```
{
  "alg": "RS256",      //Signing Algorithm is RS256
```

```
    "kid": "0001",        //Put your own Key ID value, "0001" is just an example
    "iat": "1625587913"   //Issued At - the time this request is sent, in Unix Time format
  }
```

3. Create **JWS Object** by combining JWS Header and Message Payload
4. Retrieve your **Private Key** as the signer
5. Create **Signed JWS Object** by signing it with the Private Key

Next, you are going to **Encrypt** the Signed JWS Object:

```
private JWEObject getEncryptedJWEObject(JWSObject jwsObject, RSAPublicKey key)
  throws JOSEException {
#1  Payload jwepayload = new Payload(jwsObject.serialize());

#2  JWEHeader jweheader = new JWEHeader.Builder(JWEAlgorithm.RSA_OAEP_256, EncryptionMethod.A128GCM)
#3  JWEObject jweObject = new JWEObject(jweheader, jwepayload);

#4  JWEEncrypter encrypter = new RSAEncrypter(key);
#5  jweObject.encrypt(encrypter);

    return jweObject;
}
```

1. Prepare your **JWE Payload**, that is, the `Signed JWS Object`
2. Create **JWE Header**. The algorithm used to encrypt the message body is `A128GCM` while the algorithm used to encrypt the encryption key is `RSA_OAEP_256`. **JWE keyID** is `0002`.
3. Create **JWE Object** by combining JWE Header and JWE Payload
4. Retrieve **HSBC's Public Key** as the encrypter
5. Create **Encrypted JWE Object** by encrypted it with HSBC's Public Key

Yes, you are now ready to put the Encrypted JWE Object as the message body *(you may need to first serialize it into String format, depends on your program code design)* of any API call.

How to Decrypt Message and Verify Signature of an Incoming Message

Every message sent from HSBC must be decrypted and verified. From the point of view of a Merchant, an **Incoming Message** means:

- the Respond Message of a Normal API, or
- the Request Message of a Callback API.

Let's look into the following example to see how you decrypt a response message from HSBC:

```
private String decryptMessage(String respMsgPayload, KeyStoreFactory keyStore)
  throws KeyStoreException, NoSuchAlgorithmException, CertificateException, IOException,
         java.text.ParseException, UnrecoverableKeyException, JOSEException {
#1  JWEObject jweObject = JWEObject.parse(respMsgPayload);

#2  PrivateKey privateKey = (PrivateKey) keyStore.getPrivateKey("merchant_private_key_alias");

    JWEDecrypter decrypter = new RSADecrypter(privateKey);
#3  jweObject.decrypt(decrypter);

#4  String signedMessage = jweObject.getPayload().toString();
    return signedMessage;
}
```

1. Create **Encrypted JWE Object** by parsing the encrypted response message payload
2. Retrieve **Private Key** as the decrypter
3. Decrypt the JWE Object using your Private Key
4. Get the **Signed Message** from the decrypted JWE Object

You are now able to extract the plain `json` message. Yet, before that, you **must** verify the signature to guarantee data integrity.

```
private String verifySignature(String signedMessage, KeyStore ks, String keyAlias)
  throws KeyStoreException, JOSEException, ParseException {
#1  JWSObject jwsObject = JWSObject.parse(signedMessage);

    Certificate certificate = ks.getCertificate(keyAlias);
#2  JWSVerifier verifier = new RSASSAVerifier((RSAPublicKey) certificate.getPublicKey());

#3  if (!jwsObject.verify(verifier)) {
      throw new ValidationException("Invalid Signature");
    }
#4  return jwsObject.getPayload().toString();
}
```

1. Create **JWS Object** by parsing the `Signed Message`
2. Retrieve **HSBC's Public Key** as the verifier
3. Verify the signed JWS Object. Invoke error handling if invalid signature found *(depends on your code design)*
4. Get the plain `json` message for further actions

Summary

Components \ Steps	Message Signing	Message Encryption	Message Decryption	Verify Signature
JWS Object	Signing Algorithm: `RS256`			

JWE Object		JWE Algorithm: `RSA_OAEP_256` Encryption Method: `A128GCM`		
KeyID	`0002`	`0002`		
Merchant's Private Key	Used as `Signer`		Used as `Decrypter`	
HSBC's Public Key		Used as `Encrypter`		Used as `Verifier`

How to Make API Request

API request can be submitted without Message Encryption, in case you want to:

- understand the basic API Call quick;
- test API connectivity before spending substantial development effort on Message Encryption.

However, data encryption is actually a required data security imposed by HSBC standard, Merchant has to invoke the encryption logic before moving to Production and fully tested during testing phase.

Make Your API Request with Plain Messages

> ! **NOTICE:**
> Skipping message encryption is the flexibility provided in Sandbox Environment for testing purpose.

Submit API request using cURL™ as an example
cURL™ is a simple command line tool that enables you to make any HTTP request. Merchant can choose any other GUI tool such as Postman™ and SoapUI™.

Step 1. Run this command in your system platform:

POST

```
curl -X POST "https://devclustercmb.api.p2g.netd2.hsbc.com.hk/glcm-mobilecoll-mcth
 -H "message_encrypt: false"
 -H "Authorization: Basic eW91cl91c2VybmFtZTp5b3VyX3Bhc3N3b3Jk"
 -H "x-HSBC-client-id: 8b915a4f5b5047f091f210e2232b5ced"
 -H "x-HSBC-client-secret: 1bb456a541dc416dB6016B5F9583C606"
```

```
#6  -H "x-HSBC-msg-encrypt-id: 42298549900001+0001+0002"
#7  -H "Content-Type: application/json"
#8  -d "{ \"txnRef\": \"PAY-QJZV956664\", \"merId\": \"42298549900001\"}"
```

1. Submit `POST` request to the API URL endpoint
2. Put the secret header `message_encrypt: false` to indicate this API request is without message encryption. This header is only applicable in Sandbox environment.
3. Put the Basic Authorization in HTTP header `Authorization`
4. Put Client ID in HTTP header `x-HSBC-client-id`
5. Put Client Secret in HTTP header `x-HSBC-client-secret`
6. Put Merchant ID, JWS ID and JWE ID in HTTP header `x-HSBC-msg-encrypt-id` respectively
7. Set `Content-Type` to JSON format
8. Plain `json` message payload

GET

Step 2. Receive response message in plain `json` format.

Making API Request with Message Encryption

Step 1. Run this cURL™ command in your system platform:

POST

```
#1 curl -X POST "https://devclustercmb.api.p2g.netd2.hsbc.com.hk/glcm-mobilecoll-mcth
#2  -H "Authorization: Basic eW91cl91c2VybmFtZTp5b3VyX3Bhc3N3b3Jk"
#3  -H "x-HSBC-client-id: 8b915a4f5b5047f091f210e2232b5ced"
#4  -H "x-HSBC-client-secret: 1bb456a541dc416dB6016B5F9583C606"
#5  -H "x-HSBC-msg-encrypt-id: 42298549900001+0001+0002"
#6  -H "Content-Type: application/json"
#7  -d "eyJraWQiOiIwMDAxIiwiZW5jIjoiQTEyOEdDTSIsImFsZyI6IlJTQS1PQUVQLTI1NiJ9.W4nobHoV
```

1. Submit `POST` request to the API URL endpoint. Any `{id}` adhered in the URL must be encrypted.
2. Put the Basic Authorization in HTTP header `Authorization`
3. Put Client ID in HTTP header `x-HSBC-client-id`
4. Put Client Secret in HTTP header `x-HSBC-client-secret`
5. Put Merchant ID, JWS ID and JWE ID in HTTP header `x-HSBC-msg-encrypt-id` respectively
6. Set `Content-Type` to JSON format
7. Encrypted Message Payload.

GET

! **NOTICE:**
Data Encryption invokes compulsory prerequisites, such as JOSE library and program coding, please make

> sure the section Message Security has been gone through thoroughly.

Step 2. For a successful request (HTTP Status Code 200), an encrypted response message will be returned, otherwise, a plain `json` with failure message will be returned.

Data Type Overview

Data Type Control:

Data Type	Allowed Characters	Definition & Important Notice
String *(For general field)*	Alphanumeric and Symbols	General field means field which is **NOT** a critical field. HSBC system will execute characters checking upon all string fields we received in order to tackle security vulnerability, such as Cross-site Scripting. Yet, we recommend you to try use Alphanumeric only for most cases.
String *(For critical field)*	`0-9` `a-z` `A-Z` `-` `_` `.`	Critical field is used to be either a key or search criteria in HSBC backend system and hence tight restriction is applied to the allowed characters. Moreover, the starting and ending space of the string value will be trimmed before stored in HSBC system. For example, string `" example 12 34 "` will be trimmed to `"example 12 34"`. **List of Critical Fields:** All `id`(s)
Integer	`0-9`	Instead of having Max Length check for String, integer range will be checked, e.g. `0 ≤ x ≤ 9999`

Field Mandatory Control:

Field Mandatory Type	Definition & Important Notice
Mandatory	Annotated with required tag in field definition section. Field & value must be present in the request with valid `JSON` format.
Optional	Annotated with optional tag in field definition section. If you don't want to pass fields that are optional, your handler should not pass neither empty strings `{"example":""}` nor blank value `{"example":" "}`.
	Annotated with conditional tag in field definition section.

Conditional	Required under a specific condition whose logic is always provided in the field definition if it is a Conditional Field.

Time Zone Control:

Aspect	Format	Definition & Important Notice
In Request Message	`yyyy-MM-dd'T'HH:mm:ssZ`	Time zone is expected to be `GMT+8` (Malaysia standard time). Merchant is required to perform any necessary time zone conversion before submit request if needed.
In Response Message	`yyyy-MM-dd'T'HH:mm:ss±hh:mm`	Timezone returned in `api_gw` object is generated from HSBC API Gateway which located in Cloud and hence is calculated in `GMT+0`. On the other hand, time field in `response` object will be returned together with timezone information. For more details, please read each field definition carefully.

FAQ

SSL Connection Questions

Where can I find HSBC SSL server certificates?

Merchant developer is able to export SSL server certificates that has been installed in your browser. By doing this, visit the **domain** of the corresponding API endpoint in your browser. For example, to get the SSL certificate of sandbox environment, use domain name https://devcluster.api.p2g.netd2.HSBC.com.hk/

However, **in production**, we will provide a certificate and require TLS 1.2 implementation.

Message Encryption Questions

What certificates will I need to work for Message Encryption in HSBC's sandbox and production environments?

A self-sign certificate is accepted in sandbox environment. In production, Merchant is required to procure a public certificate from an authorized Certificate Authority (CA) for IT security reason.

Javascript Object Signing and Encryption (JOSE) Framework Questions

Where can I get more information about JOSE Framework?

If you want to fully understand the framework, you can read here for more details.

Please note the url does not belongs to HSBC, use it on your own discretion. By clicking the url or website, it means you accept this terms and conditions.

Where can I download JOSE libraries for development?

For your reference, you may find the following JOSE libraries of different programming languages.

- Ruby
- Python
- PHP
- Java
- Node
- .NET

Please note those urls or websites do not belong to HSBC, use it on your own discretion. By clicking those urls or websites, it means you accept this terms and conditions.

Webhooks

What is a Webhook

Webhooks (Web Callback, HTTP Push API or Reverse API) is one way one web application can send information to another application in real-time when a specific event happens.

You can use HSBC Omni Collect Webhooks to receive notifications when a specific event occurs. When one of these events is triggered, we send an HTTP POST payload in encrypted JSON to the webhook's configured URL.

Set Up

Entity	Event	URL Set Up
Payments	• payment.success • payment.settled	Contact our support team for configuration

Exception Handling

Every event that receives a non-2xx response is considered as an event delivery failure and retry mechanism will be triggered. Up to 4 retries will be triggered in every 2 minutes. Maximum 5 calls including the 1st attempt.

Idempotency

There could be scenarios where your endpoint might receive the same webhook multiple times. This could happen as an expected behaviour such as the retry mechanism or any other exceptional behaviour such as network problem.

To handle duplicate webhooks, we offer a unique webhook ID `x-hsbc-webhook-id` where you can find it in the HTTP header on every webhook.

Webhooks

Webhooks for Payments

POST `/<Callback URL predefined by Merchant>`

DESCRIPTION

The table below lists the Webhook events available for payments.

Webhook Event	Definition
payment.success	Triggered when a payment is successfully captured.
payment.settled	Triggered when a payment is settled.

REQUEST PARAMETERS

x-hsbc-webhook-id required in header	UUID
Content-Type: string required in header	text/plain

REQUEST BODY

PaymentWebhook	*Data Encryption is enforced. API Schema intends to demonstrate the skeleton of the message payload only.*

Request Content-Types: text/plain

Request Example

```
{
  "webhook": {
    "event": "payment.success",
    "entities": [
      "payment"
    ]
  },
  "payload": {
    "payment": {
      "id": "14627849160897986",
      "sqn": "14627849160897986",
      "merchant_id": "T014570T8280000",
      "master_merchant_id": null,
      "amount": 1000,
      "currency": "THB",
      "payer_bank_code": "003",
      "status": "SUCCESS",
      "txn_datetime": "2020-01-01T13:02:00+07:00",
      "notify_datetime": "2020-01-01T13:02:00+07:00",
      "value_date": "2020-01-01",
      "type": "CREDIT_TRANSFER",
      "credit_account": "******570001",
      "proxy": {
        "id": "******642001",
        "type": "ANY_ID"
      },
      "channel": "MOBILE",
      "terminal_id": null,
      "ref1": null,
      "ref2": null,
      "ref3": null
    }
  }
}
```

RESPONSES

200 OK Callback	Successful operation. *Data Encryption is enforced. API Schema intends to demonstrate the skeleton of the message payload only.*

Response Content-Types: application/json

Response Example (200 OK)

```
{
  "status": "SUCCESS"
}
```

Schema Definitions

PaymentWebhook: object

PROPERTIES

webhook: object required

PROPERTIES

event: string enum: [payment.success, payment.settled] range: (up to 100 chars) required

Event Type

entities: string[] required

The list of Entities contained in this Webhook

ITEMS

string enum: [payment]

payload: object required

PROPERTIES

payment: Payment required

Example

```
{
  "webhook": {
    "event": "payment.success",
    "entities": [
      "payment"
    ]
  },
  "payload": {
    ...See schema Payment for details...
  }
}
```

Payment: object

PROPERTIES

id: string range: (up to 36 chars) required

Unique Entity ID of a Payment

sqn: string range: (up to 36 chars) required

A reference number for payment

merchant_id: string range: (up to 35 chars) required

Merchant ID

master_merchant_id: string range: (up to 35 chars) required

Master Merchant ID

amount: integer range: 1 = x = 999999999999 required

Payment Amount

- Format: Eliminate punctuation and sign, support 2 decimal places according to ISO 4217, e.g. $10.50 = 1050

currency: string enum: [THB] range: (up to 3 chars) required

Payment Currency

payer_bank_code: string range: (up to 3 chars) required

Payer Bank Code

status: string enum: [SUCCESS, SETTLED] range: (up to 20 chars) required

Payment Status

txn_datetime: string range: (up to 25 chars) required

Transaction time for the inward credit payment

- Bank system local time. A `GMT+7` timezone information is appended to the end of the timestamp to indicate this time is a Thailand local time. Format: `yyyy-MM-dd'T'HH:mm:ss±hh:mm`

notify_datetime: string range: (up to 25 chars) required

Notification time for the inward credit payment

- Bank system local time. A `GMT+7` timezone information is appended to the end of the timestamp to indicate this time is a Thailand local time. Format: `yyyy-MM-dd'T'HH:mm:ss±hh:mm`

value_date: string range: (up to 10 chars) required

Value Date

type: string enum: [CREDIT_TRANSFER, BILL_PAYMENT] range: (up to 30 chars) required
Payment Type

credit_account: string range: (up to 35 chars) required
Credit Account

proxy: object required

PROPERTIES

id: string range: (up to 128 chars) required
Proxy ID

type: string enum: [ANY_ID, BILLER_ID, SETTLEMENT_ACCOUNT, VIRTUAL_ACCOUNT] range: (up to 30 chars) required
Proxy Type

channel: string enum: [KIOSK, EDC_POS, INTERNET, MOBILE, OTHERS] range: (up to 30 chars) required
Channel

terminal_id: string range: (up to 16 chars) required
Terminal ID

ref1: string range: (up to 24 chars) required
Reference 1

ref2: string range: (up to 24 chars) required
Reference 2

ref3: string range: (up to 24 chars) required
Reference 3

Example

```
{
  "id": "14627849160897986",
  "sqn": "14627849160897986",
  "merchant_id": "T014570T8280000",
  "master_merchant_id": null,
  "amount": 1000,
  "currency": "THB",
  "payer_bank_code": "003",
  "status": "SUCCESS",
  "txn_datetime": "2020-01-01T13:02:00+07:00",
  "notify_datetime": "2020-01-01T13:02:00+07:00",
  "value_date": "2020-01-01",
  "type": "CREDIT_TRANSFER",
  "credit_account": "******570001",
  "proxy": {
    "id": "******642001",
    "type": "ANY_ID"
  },
  "channel": "MOBILE",
  "terminal_id": null,
  "ref1": null,
  "ref2": null,
  "ref3": null
}
```

Callback: object

PROPERTIES

status: string range: (up to 30 chars) required
Return Message

Example

```
{
  "status": "SUCCESS"
}
```

Lifecycle of Cryptographic Keys

This section highlights the Lifecycle of cryptographic keys in the following steps:

1. Generate keys pair (Private Key and Public Key Certificate)
2. ***Optional:*** *Export CSR (Certificate Signing Request) and get signed with CA (Certificate Authority)*

   > ! **DO YOU KNOW?**
   > In public key infrastructure (PKI) systems, a certificate signing request is a message sent from an applicant to a certificate authority in order to apply for a digital identity certificate. It usually contains the public key for which the certificate should be issued.

3. Exchange Certificate with HSBC
4. Key Maintenance
5. Key Renewal Process

Command line tool **Java Keytool™** is used in the demonstration. The tool can generate public key / private key pairs and store them into a Java KeyStore. The Keytool executable is distributed with the **Java SDK (or JRE)™**, so if you have an SDK installed you will also have the Keytool executable. Yet, Merchant is free to choose any other tool to generate and manage keys, such as **OpenSSL™**.

Key Generation and Certificate Exchange with HSBC

1. Create a new keys pair (Private Key and Public Key Certificate) with a new or existing Keystore.

```
keytool -genkey
    -alias merchant_key_pair
    -keyalg RSA
    -keystore merchant_keystore.jks
    -keysize 2048
    -validity 3650
    -storepass <your keystore password>
```

- **-genkey** - command to generate keys pair.
- **-alias** - define the alias name (or unique identifier) of the keys pair stored inside the keystore.
- **-keyalg** - key algorithm, it must be `RSA` regarding to HSBC standard. If `RSA` is taken, the default hashing algorithm will be `SHA-256`.
- **-keystore** - file name of the keystore. If the file already exists in your system location, the key will be created inside your existing keystore, otherwise, a new keystore with the defined name will be created.

> ! **DO YOU KNOW?**
> Keystore is a password-protected repository of keys and certificates. File with extension `jks` means it is a Java Keystore which is originally supported and executable with Java™.
>
> There are several keystore formats in the industry like `PKCS12` with file extension `p12` which is executable with Microsoft Windows™, merchant can always pick the one most fit their application.

- **-keysize** - key size, it must be `2048` regarding to HSBC standard.
- **-validity** - the validity period of the private key and its associated certificate. The unit is `day`, 3650 means 10 years.
- **-storepass** - password of the keystore.

1.1. Provide `Distinguished Name` information after running the command:

```
Information required for CSR generation
-------------------------------------------------------------
What is your first and last name?
  [Unknown]:  MERCHANT INFO
What is the name of your organizational unit?
  [Unknown]:  MERCHANT INFO
What is the name of your organization?
  [Unknown]:  MERCHANT INFO
What is the name of your City or Locality?
```

```
  [Unknown]:  HK
What is the name of your State or Province?
  [Unknown]:  HK
What is the two-letter country code for this unit?
  [Unknown]:  HK
Is CN=XXX, OU=XXX, O=XXX, L=HK, ST=HK, C=HK correct? (type "yes" or "no")
  [no]:  yes

Enter key password for <merchant_key_pair>
        (RETURN if same as keystore password):
Re-enter new password:
```

> ! **NOTICE:** Private Key password and Keystore password can be the same or Merchant can set them differently to be more secure.

2. **Optional:** Export CSR and get signed with CA. This step can be skipped if Merchant decides to work with a Self-Signed Certificate.

```
keytool -certreq
    -alias merchant_key_pair
    -keyalg RSA
    -file merchant_csr.csr
    -keystore merchant_keystore.jks
```

- **-certreq** - command to generate and export CSR.
- **-alias** - the name of the associated keys pair.
- **-keyalg** - key algorithm, it must be `RSA` regarding to HSBC standard.
- **-file** - file name of the CSR. This will be generated at the location where the command is run.
- **-keystore** - specify the keystore which you are working on.

2.1. Select and purchase a plan at Certificate Authority and then submit the CSR accordingly. After a signed Certificate is issued by CA, import the Certificate back to Merchant's keystore.

```
keytool -import
    -alias merchant_signed_cert_0001
    -trustcacerts -file CA_signed_cert.p7b
    -keystore merchant_keystore.jks
```

- **-import** - command to import object into a specific keystore.
- **-alias** - define the alias name (or unique identifier) of the signed Certificate.
- **-trustcacerts -file** - specify the file name of the signed Certificate in Merchant's local file system.

> ! **NOTICE:** `PKCS#7` is one of the common formats that contains certificates and has a file extension of `.p7b` or `.p7c`. The certificate format may be varied depending on the policy of the issuing CA.

- **-keystore** - specify the keystore which you are working on.

3. Export Certificate and send to HSBC for key exchange.

> ! **DO YOU KNOW:**
> A Certificate or Public Key Certificate is an electronic document that contains a public key and additional information that prove the ownership and maintain integrity of the public key. This is essential for the sender to ensure the key is not altered by any chance during delivery.

```
keytool -export
    -alias merchant_key_pair
    -file merchant_cert_0001.cer
    -keystore merchant_keystore.jks
```

- **-export** - command to export object from a specific keystore.
- **-alias** - the name of the associated keys pair.

> ! **NOTICE:** If Merchant associates the original keys pair `merchant_key_pair`, the exported Certificate is without CA-signed, and hence, Self-Signed. However, if Merchant associates the imported Certificate `merchant_signed_cert_0001` mentioned in step #2, the exported Certificate is CA-signed.

- **-file** - specify the file name of the Certificate where the file will be exported to Merchant's local file system.

> ! **NOTICE:** The default Certificate file encoding is binary. HSBC accepts both binary and base64 encoding. To export a printable base64 encoding file, please attach an extra parameter `-rfc` in the command.
> e.g. `-file merchant_cert_0001.crt -rfc`.

- **-keystore** - specify the keystore which you are working on.

4. Import HSBC's Certificate into merchant's Keystore.

```
keytool -import
    -alias hsbc_cert_0002
    -file hsbc_cert_0002.cer
    -keystore merchant_keystore.jks
```

- **-import** - command to import object into a specific keystore.
- **-alias** - define the alias name of HSBC's Certificate in your keystore.
- **-file** - specify the file name of HSBC's Certificate in Merchant's local file system.
- **-keystore** - specify the keystore which you are working on.

5. **Optional:** List keystore objects. Merchant is suggested to verify that all required objects are properly maintained. 2 - 3 entries should be found in your Java Keystore: *(Entries may be varied if other key repository*

format is used)

Alias name	Corresponding Object	Remark
merchant_key_pair	• Merchant's Private Key • Merchant's Public Certificate (Self-Signed)	These two objects appear to be one entry in a JAVA Keystore. Merchant can still export them separately into two objects (files) on your local file system depending on your application design.
merchant_signed_cert_0001	• Merchant's Public Certificate (CA-Signed)	Not exist if Merchant skips step #2
hsbc_cert_0002	• HSBC's Public Certificate	

```
keytool -list -v -keystore merchant_keystore.jks

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 3 entries

Alias name: merchant_key_pair
Creation date: Jan 1, 2020
Entry type: PrivateKeyEntry

<Other Information>

*******************************************
*******************************************

Alias name: merchant_signed_cert_0001
Creation date: Jan 1, 2020
Entry type: trustedCertEntry

<Other Information>

*******************************************
*******************************************

Alias name: hsbc_cert_0002
Creation date: Jan 1, 2020
Entry type: trustedCertEntry

<Other Information>

*******************************************
*******************************************
```

Certificates and Keys Maintenance

Here are some recommendations to Merchant of how to properly maintain certificates and keys:

Component	Storage	Validity
Merchant's Private Key	Private Key should be maintained and handled with the most secure approach that a Merchant can apply. The most common and yet secure enough approach is: • **key password** - Do not save the password in plain text or hard-coded in application. Recommend to encrypt it by any Password Encryption Tools • **key storage** - Store inside password-protected key repository, such as `JKS` or `PKCS12` keystore. Keystore password should also be encrypted.	No restriction on the Validity Period. However, if Merchant suspects there is any chance that the key is leaked or for any other security reason, a new Private Key and its associated Public Key Certificate should be generated.
Merchant's Public Key Certificate	Since Public Key Certificate is publicly distributed, a comparative moderate secure storage approach is acceptable. Merchant can store the physical file in any system's file system or store all keys and certificates in one single key repository for a centralised key management.	For a self-signed Certificate, the same condition has been mentioned as above. However, the validity period of a CA-signed Certificate is depended on the purchase plan of the issuing CA. The most common standard is 1 to 2 years.
HSBC's Public Key Certificate	Same as the above	1 Year **NOTICE:** Technically, the validity period is usually 1 Year plus 1 to 2 months more. The spare period is a buffer for a merchant to switch a "to-be-expired" Certificate to the new one during the Certificate Renewal Process. More technical detail will be covered in later section.

Certificates and Keys Renewal

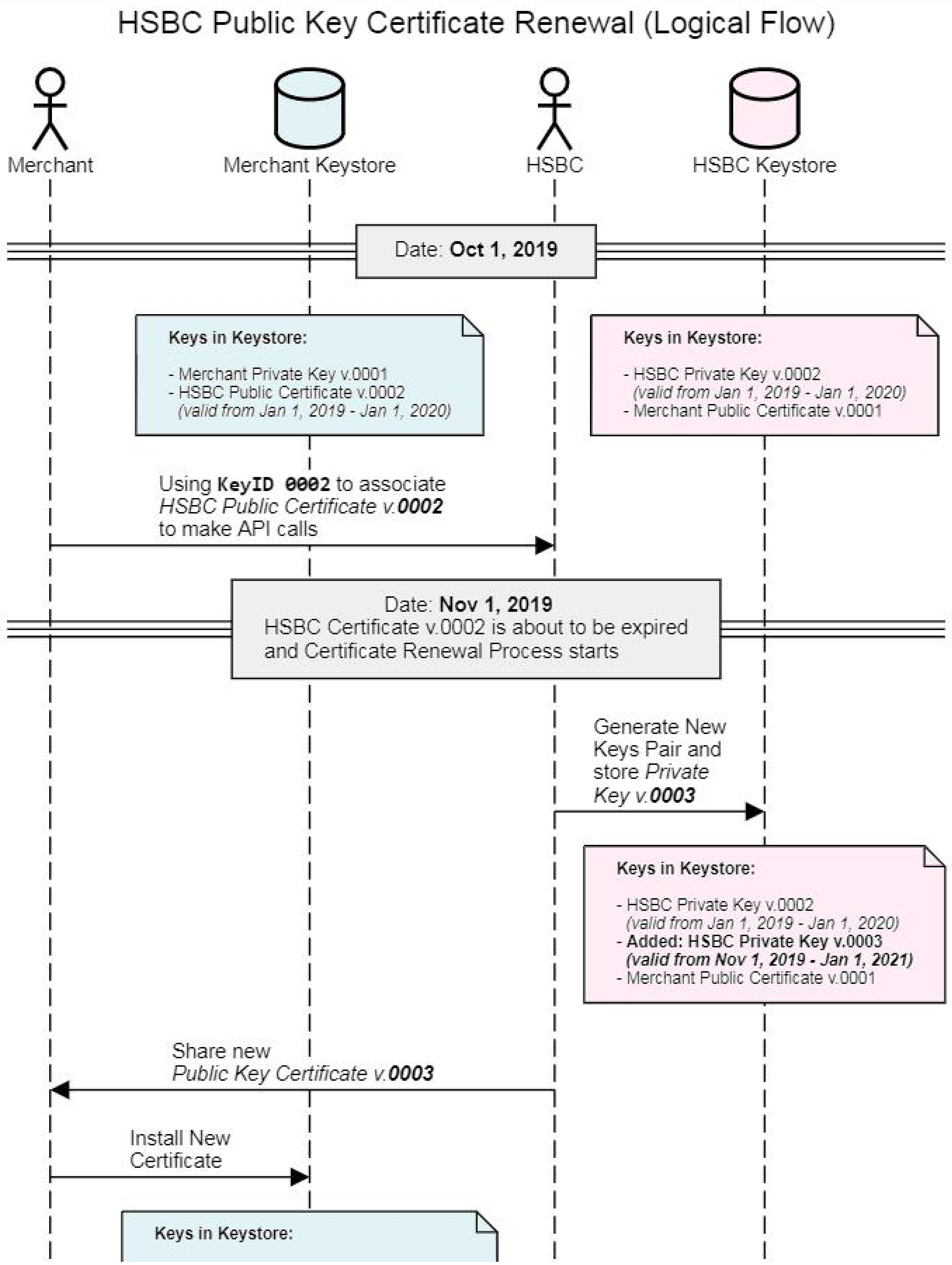
Every Public Key Certificate has an expiration date and when either Merchant's or HSBC's Certificate is about to expire, a key renewal process will be taken place. Please see the below Key Renewal Process Flow for your reference:

! **SOME RULES YOU SHOULD KNOW:**

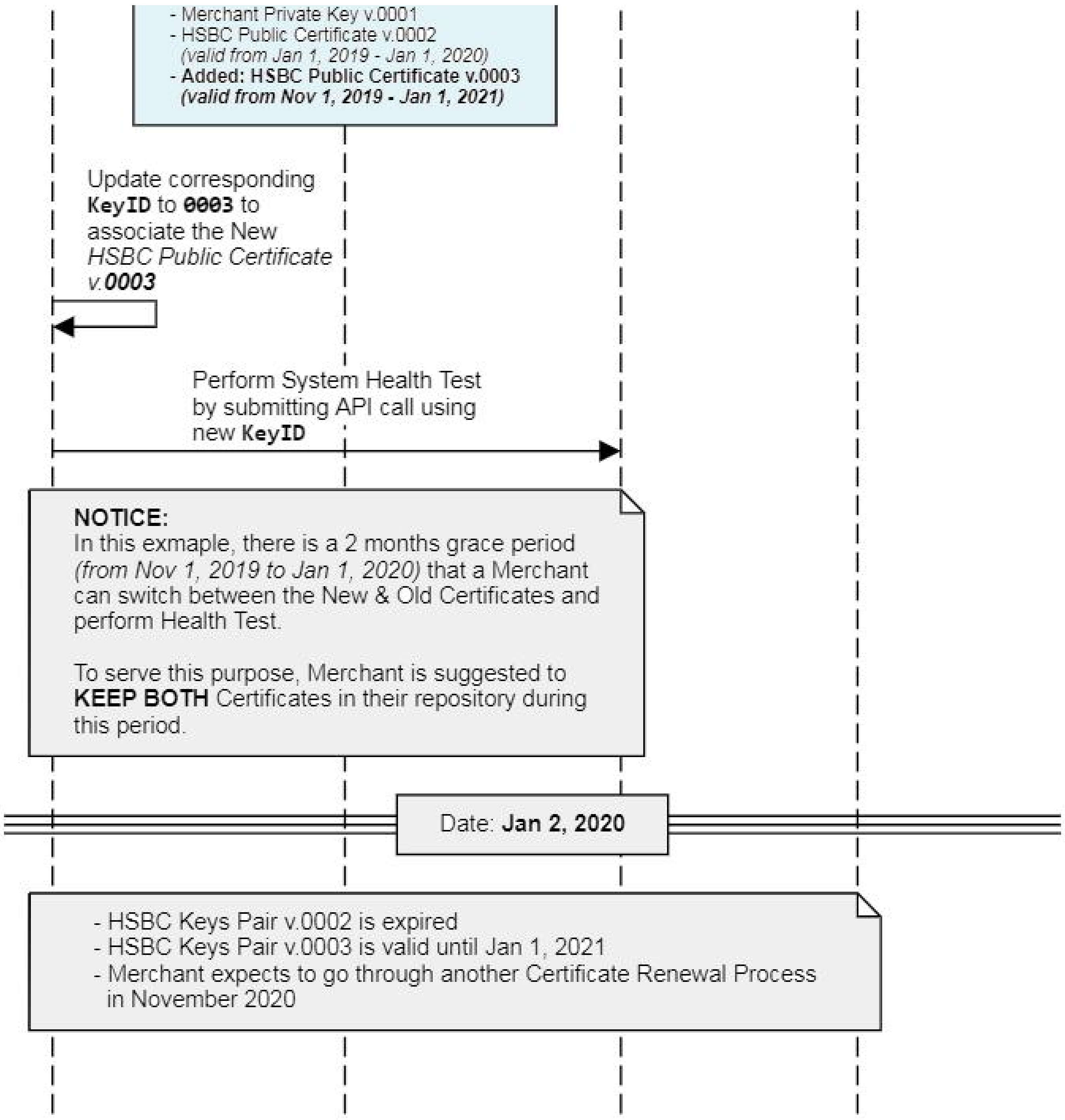
- **Keys Repository:** This is a make-up for demonstration purpose only.
- **Keys Name:** Using a `Key Name` `KeyID` naming convention is for a simpler demonstration. The suggested identifier of one key should be the alias name inside a key repository.
- **KeyID Value:** HSBC uses naming convention `0001`, `0002`, `0003`... `n + 1`, when every time HSBC certificate is renewed, the `KeyID` value will be `n + 1`.
- **KeyID Binding:** The binding between `KeyID` and corresponding `Keys Pair` in merchant's system can make use of any key/value logic, such as Database table. In our example below, KeyID `000X` binds to

`Private Key v.000X` and `Public Certificate v.000X`, etc.

- **Validity Date:** All dates are make-up for demonstration purpose only.

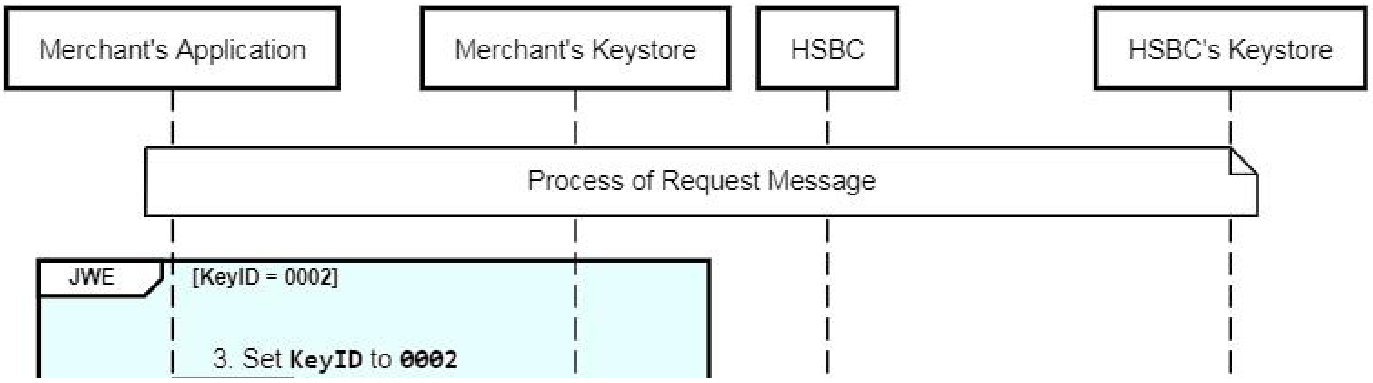


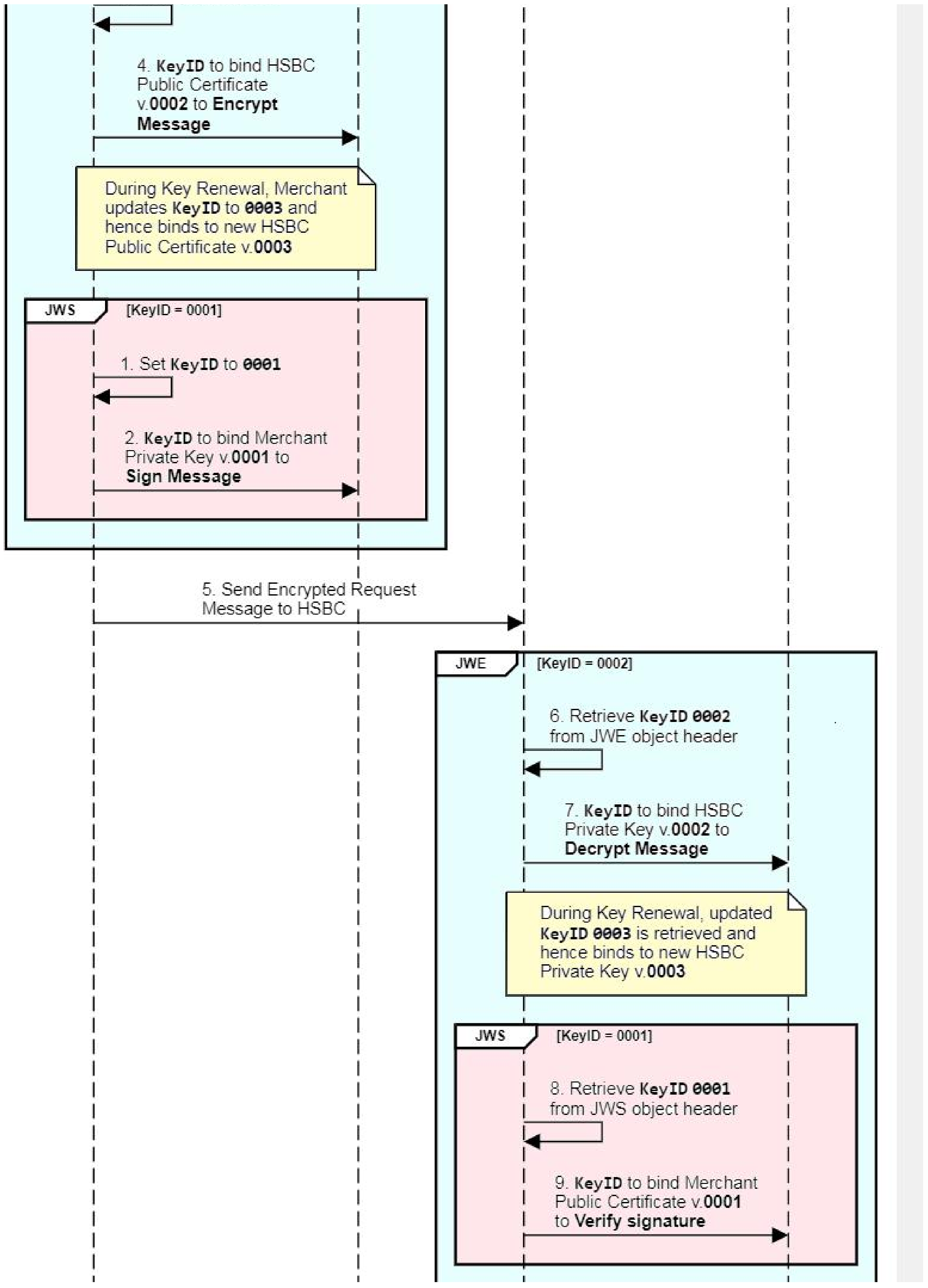
HSBC Public Key Certificate Renewal (Logical Flow)
Merchant
Merchant Keystore
HSBC
HSBC Keystore
Date: Oct 1, 2019
Keys in Keystore:
- Merchant Private Key v.0001
- HSBC Public Certificate v.0002
(valid from Jan 1, 2019 - Jan 1, 2020)
Keys in Keystore:
- HSBC Private Key v.0002
(valid from Jan 1, 2019 - Jan 1, 2020)
- Merchant Public Certificate v.0001
Using KeyID 0002 to associate HSBC Public Certificate v.0002 to make API calls
Date: Nov 1, 2019
HSBC Certificate v.0002 is about to be expired and Certificate Renewal Process starts
Generate New Keys Pair and store Private Key v.0003
Keys in Keystore:
- HSBC Private Key v.0002
(valid from Jan 1, 2019 - Jan 1, 2020)
- Added: HSBC Private Key v.0003
(valid from Nov 1, 2019 - Jan 1, 2021)
- Merchant Public Certificate v.0001
Share new Public Key Certificate v.0003
Install New Certificate
Keys in Keystore:



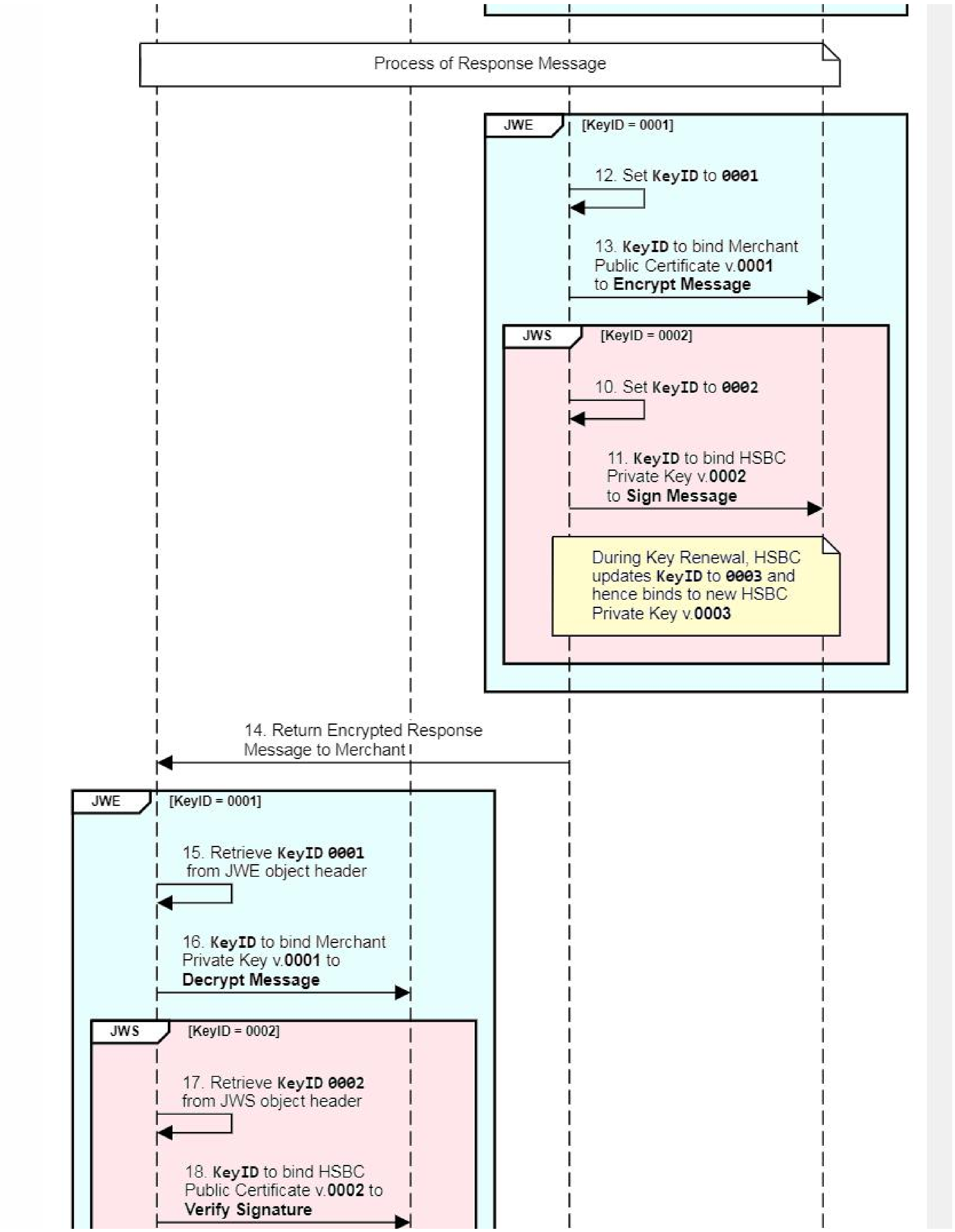
- Merchant Private Key v.0001
- HSBC Public Certificate v.0002 (valid from Jan 1, 2019 - Jan 1, 2020)
- Added: HSBC Public Certificate v.0003 (valid from Nov 1, 2019 - Jan 1, 2021)
Update corresponding KeyID to 0003 to associate the New HSBC Public Certificate v.0003
Perform System Health Test by submitting API call using new KeyID
NOTICE: In this exmaple, there is a 2 months grace period (from Nov 1, 2019 to Jan 1, 2020) that a Merchant can switch between the New & Old Certificates and perform Health Test.
To serve this purpose, Merchant is suggested to KEEP BOTH Certificates in their repository during this period.
Date: Jan 2, 2020
- HSBC Keys Pair v.0002 is expired
- HSBC Keys Pair v.0003 is valid until Jan 1, 2021
- Merchant expects to go through another Certificate Renewal Process in November 2020

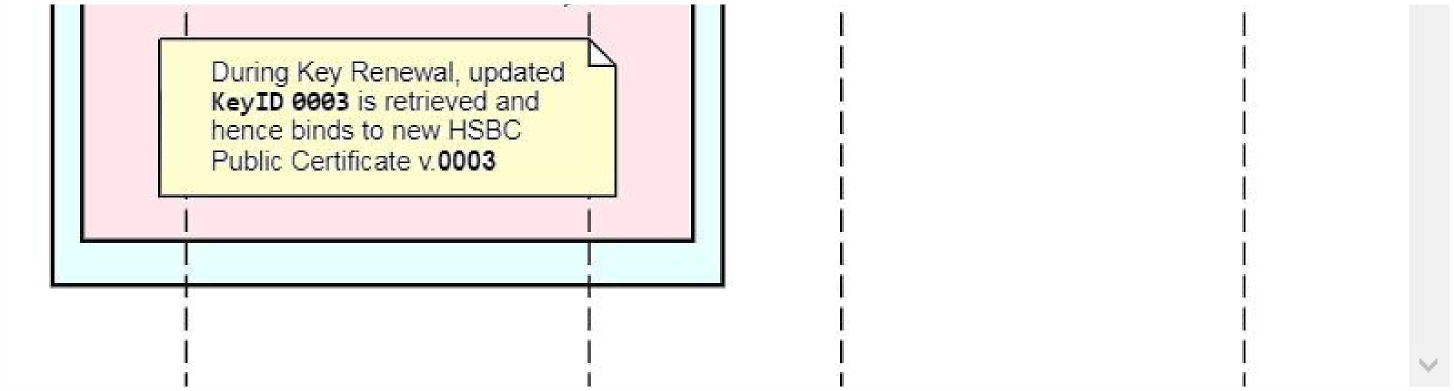
Below is the technical flow showing how `Certificates`, `Alias Names` and `KeyIDs` work together during a normal process or a key renewal process:





4. KeyID to bind HSBC Public Certificate v.0002 to Encrypt Message
During Key Renewal, Merchant updates KeyID to 0003 and hence binds to new HSBC Public Certificate v.0003
JWS
[KeyID = 0001]
1. Set KeyID to 0001
2. KeyID to bind Merchant Private Key v.0001 to Sign Message
5. Send Encrypted Request Message to HSBC
JWE
[KeyID = 0002]
6. Retrieve KeyID 0002 from JWE object header
7. KeyID to bind HSBC Private Key v.0002 to Decrypt Message
During Key Renewal, updated KeyID 0003 is retrieved and hence binds to new HSBC Private Key v.0003
JWS
[KeyID = 0001]
8. Retrieve KeyID 0001 from JWS object header
9. KeyID to bind Merchant Public Certificate v.0001 to Verify signature


Process of Response Message
JWE [KeyID = 0001]
12. Set KeyID to 0001
13. KeyID to bind Merchant Public Certificate v.0001 to Encrypt Message
JWS [KeyID = 0002]
10. Set KeyID to 0002
11. KeyID to bind HSBC Private Key v.0002 to Sign Message
During Key Renewal, HSBC updates KeyID to 0003 and hence binds to new HSBC Private Key v.0003
14. Return Encrypted Response Message to Merchant
JWE [KeyID = 0001]
15. Retrieve KeyID 0001 from JWE object header
16. KeyID to bind Merchant Private Key v.0001 to Decrypt Message
JWS [KeyID = 0002]
17. Retrieve KeyID 0002 from JWS object header
18. KeyID to bind HSBC Public Certificate v.0002 to Verify Signature


During Key Renewal, updated KeyID 0003 is retrieved and hence binds to new HSBC Public Certificate v.0003

! **NOTICE:** All examples above are about the Certificate Renewal of HSBC, whenever Merchant wants to renew their Certificate, please switch your role and steps into HSBC's.

Disclaimer

IMPORTANT NOTICE

This document is issued by The Hongkong and Shanghai Banking Corporation Limited, Hong Kong (“HSBC”). HSBC does not warrant that the contents of this document are accurate, sufficient or relevant for the recipient’s purposes and HSBC gives no undertaking and is under no obligation to provide the recipient with access to any additional information or to update all or any part of the contents of this document or to correct any inaccuracies in it which may become apparent. Receipt of this document in whole or in part shall not constitute an offer, invitation or inducement to contract. The recipient is solely responsible for making its own independent appraisal of the products, services and other content referred to in this document. This document should be read in its entirety and should not be photocopied, reproduced, distributed or disclosed in whole or in part to any other person without the prior written consent of the relevant HSBC group member. Copyright: HSBC Group 2019. ALL RIGHTS RESERVED.