Connectivity Guide

eftpos API Gateway

eftpos

Good for Australia

apisupport@eftposaustralia.com.au

last updated: 09-Dec-2022

Copyright, confidentially and disclaimer

Table of Contents

1. Connecting to the eftpos API Gateway

To successfully connect to the eftpos API Gateway services the following connectivity is required:

1. *Transport Layer Security* - Mutual Transport Layer Security (mTLS) Authentication v1.2 is used for secure connection between systems for PROD and CERT testing.
2. *Application Layer Security* - OAuth 2.0 Client Credentials Grant is used to provide access to different API Products to those only authorised API Clients.
3. *Message Layer Security* – dependent on the backend service, e.g., the eftpos Tokenisation Service uses JWE for passing sensitive card and token information within the request and response to/from the eTSP.

1.1. eftpos API environments

Environment	**Host**	**mTLS (v1.2)**
1. PROD (SECURE)	**sapi**.eftpos.io/{api-path}	Yes
2. CERT (SECURE)	**cert.sapi**.eftpos.io/stable/{api-path}	Yes
3. SANDBOX (SECURE)	**sandbox.sapi**.eftpos.io/{api-path}	Yes
4. SANDBOX	**sandbox.api**.eftpos.io/{api-path}	No

1.2. eftpos APIs and Paths

eftpos API Products	API Path	Message Layer Security
1. BIN Inquiry API	/bin/v1	Not Required
2. CNP Payment API	/payment/v2 /payment/v3	Not Required
3. Fast Funds API	/fastfunds/v1	Not Required
4. Token on File API	/ets/v1	JWE
5. Disputes & Chargebacks Create (Issuer) API	/issuer/v1	Not Required
6. Disputes & Chargebacks (Update/Acquirer) API	/dispute/v1	Not Required
7. Reference Data API	/info/v1	Not Required
8. QR Order API	/qrorder/v1	HMAC
9. QR Wallet API	/qrcode/v1	HMAC

1.3. API Security

1.3.1. Transport Layer Security

For the initial set up and configuration of Mutual Transport Layer Security (mTLS) v1.2 it is necessary to generate a private key and a certificate signing request (CSR) using the command: -

```
openssl req -new -config csr.cnf -keyout key.pem -out certificate.csr
```

Where the following information is contained in the `csr.cnf` file: -

`fqdn` (fully qualified domain name) must be of the format

- nonprod.<YOUR_ORG_NAME>.<YOUR_APP_Name_OPTIONAL>.api.eftpos.io for SANDBOX and CERT environment
- <YOUR_ORG_NAME>.<YOUR_APP_Name_OPTIONAL>.api.eftpos.io for PROD environment

orgName

- Should be a name that is representative of your company.

appName

- Should be the name of the application/system/team of your company,which is connecting to the eftpos APIs.
- email must be
- A team or group email, i.e., a persistent email address. Do not use an individual employee's email address or email address that is not likely to be in place in the longer term.

fqdn Examples:

Without app name:

- nonprod.fisglobal.api.eftpos.io (For Sandbox/Cert)
- fisglobal.api.eftpos.io (For production)

With app name:

- nonprod.fisglobal.dn.api.eftpos.io (For Sandbox/Cert)
- nonprod.fisglobal.ist.api.eftpos.io (For Sandbox/Cert)
- fisglobal.api.dn.eftpos.io (For production)
- fisglobal.api.ist.eftpos.io (For production)

The full template for the CSR file is as follows

```
# OpenSSL configuration file for creating a CSR for a server certificate
# Adapt at least the fqdn and orgName lines, and then run
# openssl req -new -config csr.cnf -keyout key.pem -out certificate.csr
# on the command line.

# The fully qualified server (or service) name.
# Omit the "nonprod." prefix for production requests.


fqdn = nonprod.<ORGANISATION_NAME>.<APPLICATION_Name_OPTIONAL>.api.eftpos.io

# The name of the organization that will own this client certificate
orgName = <ORGANISATION_NAME>

# The email address to send the signed certificate to. This should ideally be a
distribution
# list to minimise impact during certificate rotations. A near-expiry reminder email
will
# also be sent to this address.
email = <ORGANISATION_EMAIL_DISTRIBUTION_LIST>

# subjectAltName entries: to add DNS aliases to the CSR.
# This parameter is only be necessary when issuing a server certificate, but it
doesn't hurt to include it.
# Delete the '#' character in the ALTNAMES line (to comment it out), and change the
subsequent
# 'DNS:' entries accordingly. Please note: all DNS names must resolve to the same IP
address as the FQDN.
altNames = DNS:$fqdn   # , DNS:bar.example.org , DNS:www.foo.example.org

# --- no modifications required below ---
[ req ]
default_bits = 2048
default_md = sha256
prompt = no
encrypt_key = no
distinguished_name = dn
req_extensions = req_ext

[ dn ]
countryName = AU
stateOrProvinceName = NSW
localityName = Sydney
organizationName = $orgName
commonName = $fqdn
emailAddress = $email

[ req_ext ]
subjectAltName = $altNames
```

Once the command is executed, this should generate a private key file named key.pem and a CSR file named certificate.csr.

- Keep your private key in a secure location and send only the certificate.csr file to the following email Ids for review.

  To: ems@eftposaustralia.com.au

  Cc: apisupport@eftposaustralia.com.au ,eftpos onboarding manager email id (If known)

1.3.2. Application Layer Security

Authorization to call an eftpos API follows the standard OAuth 2.0 Client Credentials Grant.

An access token must be obtained, at least every 60minutes, by the API Client (app) authenticating against the eftpos authorisation server's token URL endpoint using its unique Client Credentials. The Client Credentials consist of a ClientID (referred to as a "Key" in the eftpos API Developer Portal) and client secret which are obtained when creating an app in the eftpos API Developer Portal.

The following table shows how client credentials are obtained from eftpos for each environment and the URL that should be used to obtain an access token.

Access Token endpoints

Environment		Access Token URL	OAuth 2.0 Client Credentials
1.	PROD	https://sapi.eftpos.io/oauth/v1/token (mTLS)	eftpos supply via the Dev Portal.
2.	CERT	https://cert.sapi.eftpos.io/oauth/v1/token (mTLS)	eftpos supply upon request
3.	SANDBOX	https://sandbox.sapi.eftpos.io/oauth/v1/token (mTLS) https://sandbox.api.eftpos.io/oauth/v1/token (No mTLS)	Self-service from Dev Portal

The eftpos API Gateway uses a Bearer Token for authorization, i.e. the following is included in the header of the POST request to obtain a token

```
Authorization: Basic <credentials>
```

Where `<credentials>` is the base64 url encoding of the concatenated string of

`<client_id>` + `:` + `<client_secret>`

And the Request body contains the the `client_id` and `grant_type` as `x-www-form-urlencoded` content type (as shown in the following cURL) example

```
curl --location --request POST
'https://sandbox.api.eftpos.io/oauth/v1/token' \
--header 'Authorization: Basic
aFNpM09YaDZacEtyb2FteDhlQ0VjNUdrUkoyZUh4T0g6SG9YR0FNYUNXbXQ0akVSdA==' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'client_id=hSi3OXh6ZpKrtalx8eCEc5GkRJ2eHxOH' \
--data-urlencode 'grant_type=client_credentials'
```

If everything is valid the `access_token` will be returned in the response as follows:-

```
{
     "client_id", "<client_id>",
     "access_token": "<access_token>",
     "expires_in": "3599",
     "scopes": "",
     "token_type": "Bearer"
}
```

The `access_token` may then be used in the header of an API request as a Bearer Token

```
Authorization: Bearer <access_token>
```

The following is a cURL example of an API call:-

```
curl --location --request GET
'https://sandbox.api.eftpos.io/stable/bin/v1/bins' \

--header 'Authorization: Bearer 2vxXgYNh0SOdWdXz5YEz7htVT1XV'
```

1.3.3. Message Layer Security

The backend service for some APIs may also require a level of security between the API Client. The type of security is specific to the backend service and hence where required it is listed as a separate sub section.

eQR Platform

The eQR Platform uses HMAC SHA256 to ensure integrity of the message between the API Client and eftpos. The following information must be passed in the header

Header	Example Value (used in Sandbox)
merchantReferenceId (/qrorder APIs)	MIDBAT123456789 [with shared secret = mysecret]
walletReferenceId (/qrcode APIs)	EFTPOS [with shared secret = mysecret]
x-eqr-date	{{your value}}
x-eqr-host	{{your value}}
x-eqr-content-sha256	Calculated HMAC as per code example below using the Request Body

The following example shows the prescript (written in javascript) that is used in the developer examples in Postman to calculate the HMAC; i.e. the value for `x-eqr-content-sha256`. Simliar code will be required in your application.

```javascript
var Property = require('postman-collection').Property;
const uuid = require('uuid');
const URL = require('url');

const date = new Date().toISOString();
const referenceValue = uuid.v4().replace(/-/g, "");
pm.environment.set("randomBankAccount", referenceValue);


const resolvedBody = Property.replaceSubstitutions(pm.request.body.raw, pm.variables.toObject())
const body = resolvedBody ? JSON.stringify( JSON.parse(resolvedBody)): "";

const resolvedHeaders = Property.replaceSubstitutions(pm.request.headers, pm.variables.toObject());

const url = postman.getEnvironmentVariable("url");
const formedURL = `${url}/${pm.request.url.path.join("/")}`;
const resolvedURL =  Property.replaceSubstitutions(formedURL, pm.variables.toObject())

const generateHmac = (input, secret )  => {
  const sha = CryptoJS.SHA256(input.body);
  const sha256ForBody = CryptoJS.SHA256(input.body).toString(CryptoJS.enc.Base64);
  //date;host;content-sha256
  const signedHeaderValues = `${input.date};${input.host};${sha256ForBody}`;
  //HTTP_METHOD + '\n' + path_and_query + '\n' + signed_headers_values
  const stringToSign = `${input.httpMethod}\n${input.pathAndQueryParams}\n${signedHeaderValues}`;
  console.log(`stringToSign :${stringToSign}`);

  return CryptoJS.HmacSHA256(stringToSign, secret).toString(CryptoJS.enc.Base64);
};

function getAuthenticationHeader(body, url, method, headers, secret) {
    const generateHmacInput = {
        "host": postman.getEnvironmentVariable("eqrHost"),
        "pathAndQueryParams": url.path,
        "httpMethod": method,
        "body": body,
        "date": date
    }
    const hmacSignature = generateHmac(generateHmacInput, secret);
    return `HMAC-256 SignedHeaders=x-eqr-date;x-eqr-host;x-eqr-content-
sha256&Signature=${hmacSignature}`;
}

function addSHA256Header(body) {
    const sha256ForBody = CryptoJS.SHA256(body).toString(CryptoJS.enc.Base64);
    pm.request.headers.add({
        key: "x-eqr-content-sha256",
        value: sha256ForBody
    });
};

function addHmacAuthHeader(body) {
    const secret = postman.getEnvironmentVariable("eqrHMACSecret");
    const authHeaderValue = getAuthenticationHeader(body, URL.parse(resolvedURL), pm.request.method,
resolvedHeaders, secret );
    pm.request.headers.add({
        key: "x-hmac-authorization",
        value: authHeaderValue
    });
}
```

```
function addDateHeader() {
    pm.request.headers.add({
        key: "x-eqr-date",
        value: date
    });
}

function addHostHeader() {
    pm.request.headers.add({
        key: "x-eqr-host",
        value: postman.getEnvironmentVariable("eqrHost")
    });
}

addSHA256Header(body);
addHmacAuthHeader(body);
addDateHeader();
addHostHeader();
```

Token on File API

JSON Web Encryption (JWE) is used to protect sensitive fields. The complete message is not encrypted, only the fields indicated in API definitions will be encrypted. This constitutes an additional security layer on top of the transport layer security mechanisms.

To use JWE, the Token Requestor must generate a certificate and share it with eftpos to be configured on the eTSP. The Token Requestor's mTLS/SSL client certificate may satisfy this requirement.

The eTSP shall generate a certificate dedicated to JWE with each Token Requestor it communicates with. The certificates shall be exchanged in PEM encoded X509 format (base64url encoding).

The JSON Web Encryption may be configured to use either Compact or JSON Serialization and is broken into five pieces as per the JWE specification:

- **JWE Protected Header**: Encoded JSON string with information regarding the cryptography used for the remaining sections.
    - `alg` - algorithm: Algorithm used to encrypt the Content Encryption Key (CEK). Currently, RSA1_5 is an acceptable algorithm.
    - `enc` - encryption: Algorithm used to encrypt the content and protected header. Currently, A128CBC_HS256 is an acceptable algorithm.

- **JWE Encrypted Key**: A random value known as the Content Encryption Key (CEK). It is used to encrypt the JSON value and create the JWE Cipher Text. The Content Encryption Key is RSA encrypted and then encoded. For requests, the public key is used to encrypt the key. For responses, the private key for your Service or Organization is used to decrypt this value.

- **JWE Initialization Vector**: A random value to use as the initialization vector. It will be used encrypt the JSON value and create the JWE Cipher Text. The initialization vector is base64url encoded.

- **JWE Cipher Text**: A block of Encrypted and encoded content. The data is encrypted with the algorithm specified in the header with the Content Encryption Key and Initialization Vector.

- **JWE Authentication Tag**: The encrypted and encoded content of the JWE Protected Header. The data is encrypted with the algorithm specified in the header with the Content Encryption Key and Initialization Vector.

For more details about the JWE standard, please refer to the Internet Engineering Task Force RFC7516 document (ISSN: 2070-1721).

2. API Status Codes

Range	Code	Description	Notes
2xx - success	200	Ok	The request has succeeded.
	201	Created	The request has been fulfilled and resulted in a new resource being created.
4xx - client error	400	Bad request	The request could not be understood by the server due to malformed syntax. The client should not repeat the request without modifications.
	401	Unauthorised	Missing or invalid authentication token.
	402	Request Failed	The request syntax was well formed, but the backend service has failed to carry out the request. Typically used when financial transaction are declined for financial reasons (e.g. insufficient funds) or failed tokenization attempts.
	403	Forbidden	The server understood the request but is refusing to fulfill it. Authorisation will not help, and the request should not be repeated.
	404	Not found	The server has not found anything matching the request URI. No indication is given of whether the condition is temporary or permanent.
	405	Method not allowed	The method specified in the request is not allowed for the resource identified by the request URI.
	409	Conflict	The request could not be completed due to a conflict with the current state of the resource.
	429	Too many requests	The user has sent too many requests in a given amount of time.
5xx - server error	500	Internal server error	The server encountered an unexpected condition which prevented it from fulfilling the request.
	503	Service Unavailable	The server cannot handle the request (because it is overloaded or down for maintenance). Generally, this is a temporary state.; e.g. in Tokenization.

	504	Gateway Timeout	The server was acting as a gateway or proxy and did not receive a timely response from the upstream serve

3. API Breaking Change Policy

3.1. Overview

As APIs evolve reasonable efforts will be made to notify consumers of breaking changes and to provide consumers with reasonable time to adopt those changes.

A **breaking change** is defined as a change which may require a consumer to make complimentary changes to their own applications to avoid disruption.

A **non-breaking change** is defined as a change for which it is reasonably expected that a consumer will not need to make complimentary changes to their own applications. Consumers may adapt to non-breaking changes at their own discretion and at their own pace without undue risk of disruption.

The general approach when introducing a breaking change to an API is to:

Implement the change to a new version of the API with an incremented major version number in the URI. For example, a breaking change to the version 1 API represent by URI sapi.eftpios.io/epic/v1/epics would result in a version 2 URI sapi.eftpos.io/epic/v2/epics being created.

Advise consumers of the “old” version of the API that it is depreciated and of the date after which it will no longer be available.

Support both old and new versions of the API concurrently for a reasonable period of time to allow consumers to assess impact and make any necessary complimentary changes to their own applications.

Delete the old version of the API after the advertised date.

Variations to the general approach may be required on a case-by-case basis.

3.2. API versioning

Every API contains a major version number as a component of the URI. The version number is specified using “vN” notation. For example, “v1” in the URI sapi.eftpos.io/epic/v1/epics indicates version 1 of that API.

A major version number is always a positive whole number.

The use of the term “major version number” is taken from semantic versioning. However, unlike traditional semantic versioning implementations, minor or patch version numbers are not exposed.

Version numbers are not included in HTTP headers.

3.3. Breaking vs. non-breaking changes

In the context of the following descriptions, the term “field” may be interpreted as either a single name-value pair or as an object containing a set of name-value pairs.

3.4. Breaking change inclusions

The following shall be considered breaking changes:

Endpoints

- Deletion of an HTTP method.

Request body

- Addition of a new required field without a default value.
- Deletion of a field.
- Deletion of a value in an existing enumerated list field.

Response body

- Deletion of a field.
- Addition, Deletion or Update to supported values for an existing field where a definitive list of values has previously been specified.

Request header

- Addition of a new required request header.

Response header

- Deletion of a non-redundant response header.

Other

- Update the type of a field.
- Addition of a new validation if a pass or fail of that new validation changes the logic of the API. *For example a new validation which changed whether a request was accepted shall be considered a breaking change. Alternately, a new*

validation which only logged an internal warning event if an unexpected value were present in a field should not be a breaking change.

- Update of an existing validation if a pass or fail of that new validation changes the logic of the API. *See above.*

3.5. Non-breaking change inclusions

Non-breaking changes may be communicated after they are already made. API consumers must ensure that their applications are implemented such that they can handle the following types of non-breaking change without prior notice.

The following should be considered non-breaking changes:

Endpoints

- Addition of a new endpoint.
- Addition of a new HTTP method to an existing endpoint.

Request body

- Addition of a new optional field.
- Addition of a new required field with a default value.
- Addition of a new supported value in an existing field.

Response body

- Addition of a new field.
- Addition of a new value to an existing field where no definitive list of values has previously been specified.
- Update to the value of error message strings. *Error message text is intended for human interpretation. System logic should only rely on HTTP response codes and error codes*.
- Update to the value of an error code field where the change is from incorrect value to correct value.
- Update to the order of fields.
- Update to the length of data returned in the value of a field.
- Update to the overall response length.

Request header

- Addition of a new optional request header.

Response header

Addition of a new response header.

Deletion of a redundant response header.

Other

- Update to HTTP response code where the change is from incorrect value to correct value.