Payment Execution API documentation

1. Introduction

The Payment Execution (PE) API allows clients to make credit and debit payments using Treasury accounts in their e-commerce applications. With this API, you can create deposits (known as receive orders in Fortris) and payouts (known as send orders) and receive updates about them through callback services.

The API ensures secure communication and lets you control who can view and authorize payments. Payments are processed similarly to those done manually by a Treasury operator.

---

2. Communications

The PE API is designed with strong security in four key areas:

1. **Authentication and data integrity**: Verifies that the data comes from a trusted source and hasn't been tampered with.
2. **Encryption**: Protects your data as it travels over the internet, by encoding data in a format that is not readable or understandable without the key and secret.
3. **Replay attack prevention**: Stops attackers from repeating a valid data transmission.
4. **Trusted connections**: Only approved IP addresses can access the API.

2.1. Authentication and data integrity

To use the PE API, two unique values are needed:

1. **Client Key**: This identifies your client and must be included in all requests.
2. **Secret**: Used to verify the integrity of the data in your messages and to authenticate you on each request.

Find out more about how to generate the key and secret and how they should be included on the request.

2.2. Encryption

All communications are encrypted using HTTPS to ensure secure data transfer.

2.3. Replay attack prevention

To prevent replay attacks, each request must include a unique number called a **nonce**. This number must be higher than the one used in the previous request.

2.4. Trusted connections

You need to whitelist your IP addresses with Customer Support to use the PE API. This makes sure only trusted connections can access the API.

---

3. Backward Compatibility

The PE API ensures backward compatibility, meaning old versions still work even after updates. Here's what it means:

3.1. Stable functionality

Table of Contents

These parts of the API won't change unexpectedly:

- existing request parameters
- existing response and callback fields
- required HTTP request headers

3.2. Flexible functionality

These parts of the API might be updated:

- optional request parameters
- new values for existing parameters
- optional HTTP request headers
- new fields in API responses and callbacks
- deprecated features might be removed after notice
- the order of fields in responses or callbacks

---

4. Security

4.1. Key and Secret

You need a key and a secret to use the API. The key identifies you, and the secret is used to create a signature for each request. Include these in your request headers exactly as provided.

The following table shows how each value must be added to the request header:

Header name	Value
key	key
signature	HMAC-SHA512(uri + sha256(payload), base64decoded(Secret))

Keep your key and secret safe and don't share them with anyone.

4.2. Nonce

A nonce is a unique value that helps prevent replay attacks. Each new request must have a nonce that is greater than the previous one.

---

5. Getting started

5.1. Get your secret

To start using the PE API, generate a new key and secret by following these steps:

1. Generate a pair of private and public keys:

```
openssl genrsa -out privatekey.txt 1024

openssl rsa -in privatekey.txt -pubout -out publickey.txt
```

The first command generates the private key **privatekey.txt**. The second command uses the private key to generate the public key **publickey.txt**. The file named **key.txt** will contain the base64 encoded secret that should then be used to generate HMAC

signatures.

2. Send the public key to Customer Support.
3. Customer Support will send back your encrypted secret:

```
{
  "key": "<new-client-key>",
  "encryptedSecretBase64Encoded": "<encoded-encrypted-secret>"
}
```

You'll then decode and decrypt this secret to use it for generating HMAC signatures.

4. Firstly, decode the secret using:

```
echo "<encoded-encrypted-secret>" > key.bin.enc

base64 --decode -i key.bin.enc -o key.bin
```

If you're using the older base64 version, use **base64 -d key.bin.enc > key.bin**

5. Then, decrypt it using:

```
openssl rsautl -decrypt -inkey privatekey.txt -in key.bin -out key.txt
```

5.2. Signature calculation

Each request to the PE API needs a valid signature and you must make sure to validate this signature to ensure data integrity.

A valid signature has this structure:

```
signature = HMAC-SHA512(url + sha256(payload), base64decoded(Secret))
```

Follow these steps to generate the signature:

1. Get the URL path of the operation (excluding the hostname). For example:

```
http://psp/deposits/create
```

```
String url="/deposits/create";
```

2. Create a SHA-256 hash of the payload:

```
String payload = "{\n" +
            "\t\"accountId\": \"00000000-0000-0000-0000-00000000000\",\n" +
            "\t\"reference\": \"00000000000000\",\n" +
            "\t\"callbackUrl\": \"http://localhost:0000/\",\n" +
            "\t\"expiryDate\": \"2020-01-02T03:04:05.123Z\",\n" +
            "\t\"requestedAmount\": {\n" +
            "\t\t\"amount\": 1.00,\n" +
            "\t\t\"currency\": \"USD\"\n" +
            "\t},\n" +
            "\t\"nonce\":1\n" +
            "}";

String sha256Hex = DigestUtils.sha256Hex(payload);
```

3. Combine the URL and hash, then apply HMAC-SHA512 using your secret:

```
String computedSignature = url + sha256Hex;
```

```
byte[] keyInBytes =
Base64.getDecoder().decode(secretBase64Encoded);
String signatureHeader = new
HmacUtils(HmacAlgorithms.HMAC_SHA_512,
keyInBytes).hmacHex(computedSignature);
```

Include this signature in the request headers.

5.2.1. Example

With those values and secret in base64: bXlzZWNyZXQ=`

```
sha256HexOfPayload=99ccff6cf3ceba5f571b5b6bc6592156dda97
c534af9c67635792cffded7db05
urlPlusSha256HexOfPayload=/deposit/create/99ccff6cf3ceba5f571
b5b6bc6592156dda97c534af9c67635792cffded7db05
signature=9cced59ae5987fa669f3fe0ef533df32d1e948e58014327f
95402090480e449e3faa3290f37f1ed66bd5ffd053539651826591a
7a72666809b7203c9e6eaaf18
```

5.2.2. Changes to the signature algorithm for V3

Some endpoints for V3, like "get balances" or "find deposits," use query parameters and don't have a request body. In these cases, the query string must be included in the signature calculation to ensure a secure request. Here's the formula for generating a valid signature when there's no request body:

```
signature = HMAC-SHA512(urlWithQueryString,
base64decoded(Secret))
```

For example, when calling "find deposits" at the following URL:

```
https://psphost/v3/deposits?depositIds=b9f1a951-f7f3-4dc8-878b-
cb7ec1810ad7&queryDate=2024-01-01T15:23:48.359Z
```

The `urlWithQueryString` would be:

```
/v3/deposits?depositIds=b9f1a951-f7f3-4dc8-878b-
cb7ec1810ad7&queryDate=2024-01-01T15:23:48.359Z
```

The query string should not be encoded.

For GET requests, the signature should be generated only from the `urlWithQueryString`. There is no need to calculate a hash from an empty request body, as required in the original algorithm.

The order of query parameters for signature verification will always match the order in which they appear in the HTTP request. However, if the same query parameter appears multiple times, they must be grouped together and not split by other parameters. For example:

- Incorrect: `?param1=a&param2=b&param1=c`
- Correct: `?param1=a&param1=c&param2=b` or `?param2=b&param1=a&param1=c`

For POST requests, the signature algorithm remains the same as in the old API, as described in the PSP documentation. If a query parameter is provided, use this formula:

```
signature = HMAC-SHA512(urlWithQueryString +
sha256(payload), base64decoded(Secret))
```

5.3. IP Whitelist

To receive payment updates via callbacks, you need to whitelist your IP addresses. You can send a list of IP addresses to

Customer Support.

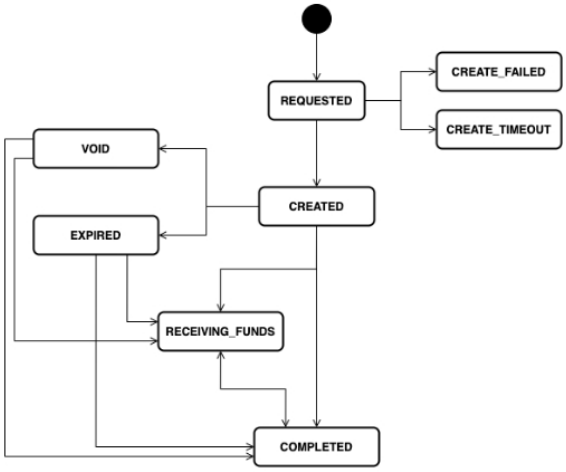
5.4. Setup PSP accounts

Ensure you have at least one valid PE account set up and activated in Treasury.

---

6. Deposits

6.1. Deposit state machine



6.2. Address verification

To enhance security during the deposit process you can use address verification. Address verification creates a digital signature of the address payload and sends it back as a part of the response.

Use the field called `verificationSignature` to ensure that the address is valid and has not been tampered with. You can find the new field in Appendix B: Callback Samples under 10.B.1 Deposits.

To validate the signature of each address, you need to:

1. Ask Customer Support for your public key to validate the signatures.
2. Build the signed payload organizationId#walletId#address.
3. Verify the signature using the algorithm SHA512withRSA.

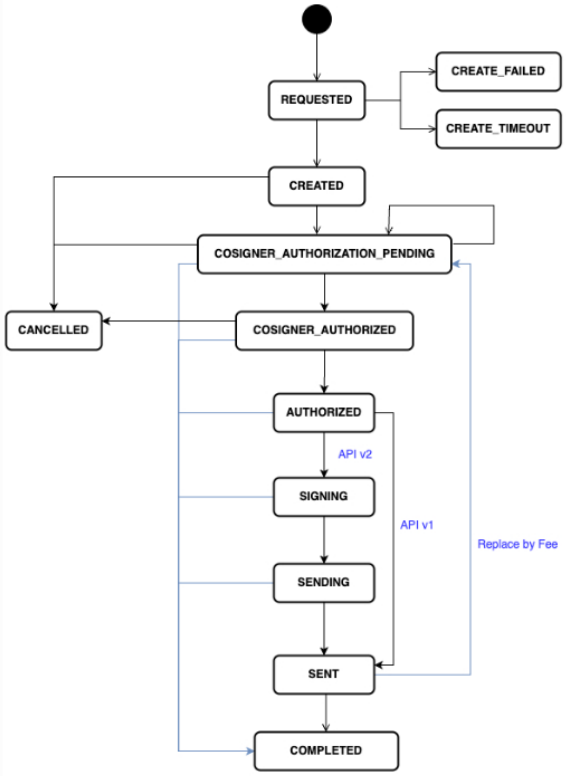
Below, you'll find an example of the verification process in Java:

```
private boolean isSignatureValid(String organizationId,
                                 String walletId,
                                 String address,
                                 String verificationSignature,
                                 String base64PublicKey) throws
GeneralSecurityException {

    var publicKey = convertFromPublicKey(base64PublicKey);
    var payloadBytes = String.join("#", organizationId, walletId,
address).getBytes(UTF_8);
    var addressSigningAlgorithm =
Signature.getInstance("SHA512withRSA");
    addressSigningAlgorithm.initVerify(publicKey);
    addressSigningAlgorithm.update(payloadBytes);
    return
addressSigningAlgorithm.verify(Base64.getDecoder().decode(verifi
cationSignature));
}

private PublicKey convertFromPublicKey(String publicKey) {
    return Try.of(() -> {
        var asymmetricKeyFactory = KeyFactory.getInstance("RSA");
        var X509publicKey = new
```

```
X509EncodedKeySpec(Base64.decodeBase64(publicKey));
        return
asymmetricKeyFactory.generatePublic(X509publicKey);
    }).get();
}
```

7. Payouts

7.1. Payout state machine



8. Account balance

The API provides an endpoint to check the balance of a PE account, available in both fiat and cryptocurrency.

8.1. Missing exchange rate

If the exchange rate isn't available, only the cryptocurrency balance will be shown.

8.2. Account balance structure

Account balances are categorized as:

- **Total balance**: All confirmed funds, plus any pending funds that are incoming or outgoing.
- **Confirmed funds**: All funds that have been confirmed and that are yours.
- **Unconfirmed funds**: All incoming funds that haven't been confirmed yet, minus all outgoing funds that haven't been confirmed yet.
- **Available balance**: The confirmed funds, minus the unconfirmed sent funds and the locked balance - these are funds that you can spend.
- **Locked funds**: Funds that are pending to be broadcast to the blockchain.
- **Unconfirmed sent balance**: Funds that have been broadcast

and are waiting to be confirmed in the blockchain.

- **Unconfirmed change balance**: Funds that are in the process of being consolidated. These are included in the available balance, as they can be spent before they are confirmed (as they come from a Fortris account instead of an external one).

9. Callbacks

Callbacks notify you about updates to your payments. Each callback includes a:

- **callback ID:** a unique identifier
- **callback type:** the type of event that triggered the callback

Callbacks ensure you receive timely updates and retry if the initial delivery fails. Validate the signature included with each callback for data integrity.

9.1. Deposit callbacks

Callbacks for deposits are sent when:

- a new deposit address is created - **DEPOSIT_CREATED**
- there are issues during deposit creation - **DEPOSIT_CREATION_FAILED**
- it takes too long to create a deposit - **DEPOSIT_CREATION_TIMEOUT**
- funds are detected but not confirmed - **DEPOSIT_RECEIVING_FUNDS**
- a user marks the corresponding receive order in Treasury as void - **DEPOSIT_VOID**
- the deposit expires in the system without any payment being made - **DEPOSIT_EXPIRED**
- payments to the address are confirmed - **DEPOSIT_COMPLETED**

9.2. Payout callbacks

Callbacks for payouts are sent when:

- a new payout is created - **PAYOUT_CREATED**
- there are issues during payout creation - **PAYOUT_CREATION_FAILED**
- the payout creation times out - **PAYOUT_CREATION_TIMEOUT**
- the payout requires further authorization - **PAYOUT_REQUIRES_AUTHORIZATIONS**
- the payout is fully authorized - **PAYOUT_COSIGNER_AUTHORIZED**
- the transaction signing process has started - **PAYOUT_SIGNING**
- the transaction broadcasting process has started - **PAYOUT_SENDING**
- payout is broadcasted to the blockchain - **PAYOUT_SENT**
- payout is confirmed - **PAYOUT_COMPLETED**
- a Treasury user cancels the associated send order - **PAYOUT_CANCELLED**

10. Appendices

Appendix A: Code samples

10.A.1. Generate a valid signature for a request

▸ **Example**

Appendix B: Callback samples

10.B.1. Deposits

▸ **Deposit created**

▸ **Deposit created with an address already used**

Fortris usually generates a new receive address for each new receive order. If you choose to save a receive address and make further payments to it, Fortris will send a callback of **DEPOSIT_COMPLETED**, but with an additional entry in the body of `duplicatedFromDepositId`.

▸ **Deposit creation failed**

▸ **Deposit creation timeout**

▸ **Deposit receiving funds**

▸ **Deposit receiving funds - multiple payments**

▸ **Deposit void**

▸ **Deposit expired**

▸ **Deposit completed**

10.B.2. Payouts

▸ **Payout created**

▸ **Payout creation failed**

▸ **Payout creation timeout**

▸ **Payout requires authorizations**

▸ **Payout cancelled**

▸ **Payout cosigner authorized**

▸ **Payout signing**

▸ **Payout sending**

▸ **Payout sent**

▸ **Payout completed**

Appendix C: Callback samples for V3

The following examples are for V3 of the PE API. All of the documentation in this guide is relevant for previous versions of the API, however the examples in Appendix B: Callback samples are specifically for Bitcoin. The V3 examples in this section are generic and can be used for any of the cryptocurrencies supported by Fortris including Bitcoin, Ethereum, Litecoin, Binance, USDT and USDC.

10.C.1. Deposits

▸ **Deposit completed in Bitcoin**

▸ **Deposit completed in Ethereum**

▸ **Deposit completed in Litecoin**

▸ **Deposit completed in USDT**

▸ **Deposit completed in USDC**

10.C.2. Payouts

▸ **Payout completed in Bitcoin**

▸ **Payout completed in Ethereum**

▸ **Payout completed in Litecoin**

▸ **Payout completed in USDT**

▸ **Payout completed in USDC**

Appendix D: Platform error codes

The most common errors are described here.

10.D.1. 4XX Client errors

401 Unauthorized

HTTP Response Code	Platform Error Code	Description
401	INVALID_HMAC_SIGNATURE	The signature header is not valid

404 Not found

HTTP Response Code	Platform Error Code	Description
404	NOT_FOUND_ACCOUNT	The Account was not found in the system
404	NOT_FOUND_XRATE	There is no Exchange Rate information for the specified currencies
404	NOT_FOUND_DEPOSIT	The Deposit was not found in the system
404	NOT_FOUND_PAYOUT	The Payout was not found in the system
404	NOT_FOUND_CLIENT_TOKEN	Client token not setup
404	NOT_FOUND_CLIENT	The key provided as header identifying the client was not found in the system

404	NOT_FOUND_ACCOUNT_BALANCE	Account balance not found for the requested accountId
404	NOT_FOUND_DEBIT_ORDER_CODE	Order code not found in the system

409 Conflict

HTTP Response Code	Platform Error Code	Description
409	CONFLICT_INVALID_NONCE	The nonce in the header was not valid
409	CONFLICT_VALIDATING_MFA_ACCESS_CODE	The OTP provided by the user was not valid
409	CONFLICT_EXPIRED_XRATE	The current Exchange Rate information in the system was expired
409	CONFLICT_PAYOUT_IN_WRONG_STATE	The Payout was not in the expected status
409	CONFLICT_PAYOUT_ALREADY_FULLY_AUTHORIRIZED	The Payout was already fully authorized
409	CONFLICT_PAYOUT_CANNOT_AUTHORIRIZE_VIA_USER	The user with username cannot authorize the Payout
409	CONFLICT_USER_ALREADY_AUTHORIZED	The user with username already provided a valid authorization
409	CONFLICT_ACCOUNT_BALANCE_NOT_ENOUGH	Not enough available balance for the operation
409	CONFLICT_CUSTOM_FEE_RATE_DISABLED	When a custom fee rate is set but the functionality is disabled

422 Unprocessable entity

HTTP Response Code	Platform Error Code	Description
422	INVALID_CURRENCY	The currency specified and the currency of the account don't match
422	INVALID_ACCOUNT	The Account was not valid
422	INVALID_DEPOSIT_STATE	The Deposit was in an invalid state
422	INVALID_PAYOUT_STATE	The Payout was in an invalid state
422	INVALID_QUERY_DATE	The queryDate was not valid. It should be in a range within the arrival time to the server +/- 2 min
422	INVALID_CUSTOM_FEE_RATE_TOO_HIGH	When a requested custom fee rate exceeds the calculated maximum rate
422	INVALID_CUSTOM_FEE_RATE_TOO_LOW	When a requested custom fee rate is lower than the minimum rate allowed

10.D.2. 5XX Server Error

500 Internal server error

HTTP Response Code	Platform Error Code	Description
500	ERROR_UNKNOWN_CREDIT_ORDER_STATE	Internal error mapping credit state to deposit state
500	ERROR_UNKNOWN_DEBIT_ORDER_STATE	Internal error mapping

		debit state to payout state
500	ERROR_RETRIEVING_TOKEN_DATA	Internal error fetching the token
500	ERROR_CALCULATING_ACCOUNT_BALANCE	Internal error calculating account balance

502 Bad gateway

HTTP Response Code	Platform Error Code	Description
502	ERROR_GETTING_CREDIT_ORDER	Internal error fetching credit information
502	ERROR_INVALID_AUTH_TOKEN_SIGNATURE	Internal error authentication
502	ERROR_IN_TOKEN_SERVICE	Internal error reading/writing the token

Last updated 2025-07-09 13:55:38 UTC