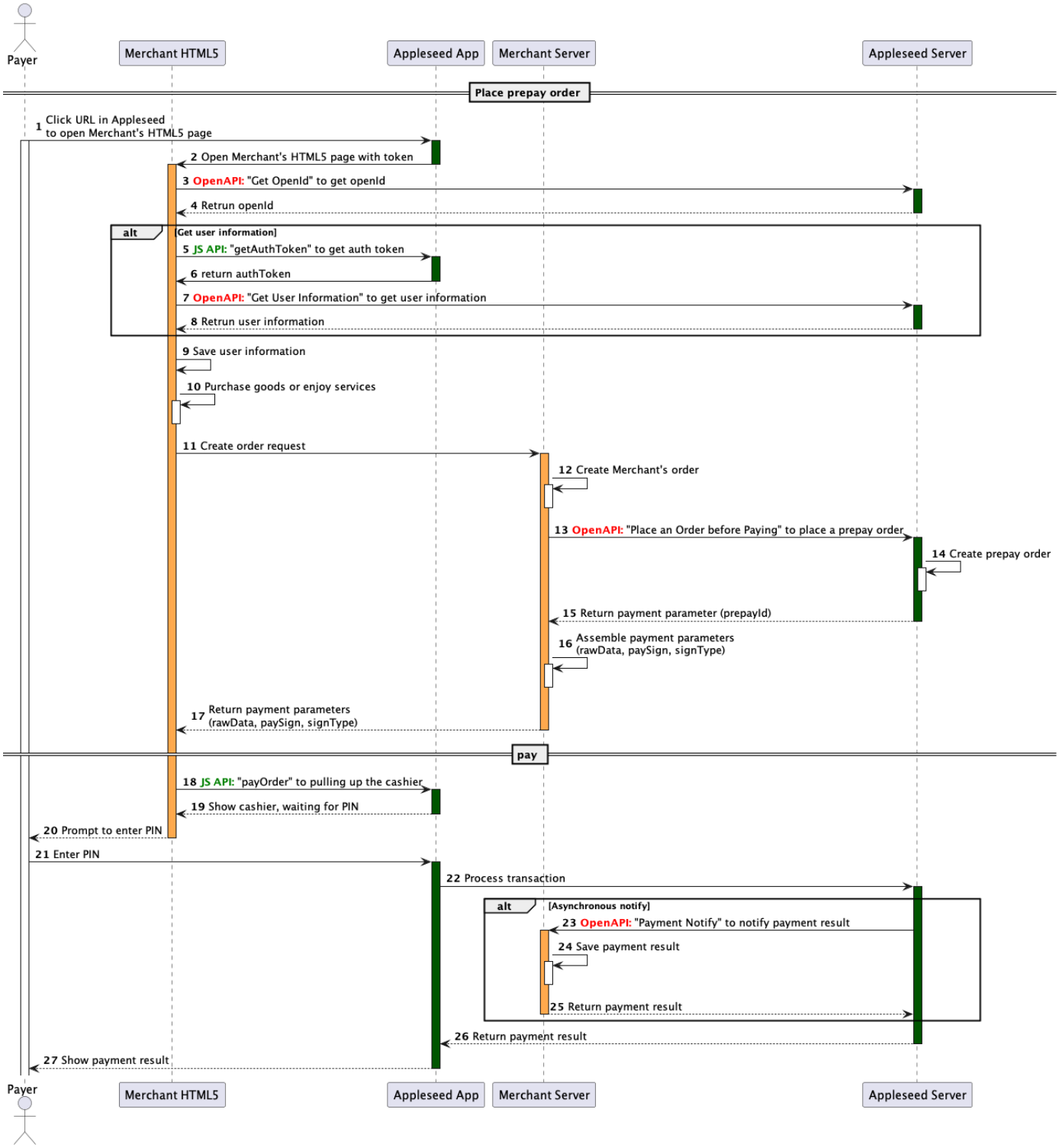
一、Development Guidelines

1 Business Sequence Chart



2 Step-by-Step Guide

2.1 Step 2: Open Merchant's HTML5 page with token

Developers need to provide the http/https url of the merchant's h5 page for the open platform to configure. When Appleseed requests the merchant's h5 url, we will deliver a token, the url such as: https://merchant.h5.com/index.html?token=xxx

2.2 Step 3: Get OpenId

Get openId via the token delivered by Step 2

Refer to the **OpenAPI** document for more request, response details, and error codes.

2.3 Step 5: Get AuthToken

If the merchant wants to get the information of Appleseed users, developers needs to get the Appleseed user authorization via JS API first.

> **Notes:** The window.payment object is not valid in other browsers.

```
const getAuthTokenRequest = {
    /// H5 application's appId
    appId: 'Appleseed_toy_shop_h5'
};

window.payment.getAuthToken(
    getAuthTokenRequest
).then((res) => {
    /// success callback
    console.log('success', res);
    var authToken = res["authToken"]; /// one-time use authorization token.
}).catch((error) => {
    /// failure callback
    console.error('fail', error);
});
```

2.4 Step 7: Get User Information

Get user information via openId returned by step 4 and authToken returned by step 6

Refer to the **OpenAPI** document for more request, response details, and error codes.

2.5 Step 13: Placing the Prepay Order

Step Description: Use this interface to submit a prepay order request for JSAPI payment, and get the `prepayId`.

Below is an example of a prepay order request:

```
{
  "mchId": "Appleseed_toy_shop",
  "appId": "Appleseed_toy_shop_h5",
  "outBizId": "2023010200010000010000023",
  "timeExpire": 1702194883,
  "description": "toy-1.00ETB",
  "callbackInfo": "callbackInfo",
  "amountCent": 100,
  "currency": "ETB"
}
```

Key parameters:

- **outBizId:** The internal order ID in the merchant's system, which is an idempotent key combined with `mchId`.
- **callbackInfo:** Information that will be sent back to the merchant's backend system when the payment result is
  notified.
- **description:** Information will be displayed on the Appleseed payment results page.

Below is an example of a prepay order response:

```
{
  "prepayId": "85711023120802000000000049007"
}
```

Refer to the **OpenAPI** document for more request, response details, and error codes.

2.6 Step 16: Assemble Payment Parameters

Step Description: After placing the prepay order, you will receive a `prepayId`. First, the developer should construct the Base String. The Base String is a concatenation of `MchId`, `AppId`, `Nonce`, `SignTimestamp`, `MchRsaSerial`, and `prepayId`, each separated by a newline character ("\n").

```
class Demo {
    String getBaseString() {
        return MchId + "\n" +
                AppId + "\n" +
                Nonce + "\n" +
                SignTimestamp + "\n" +
                MchRsaSerial + "\n" +
                PrepayId + "\n";
    }
}
```

Example:

```
mch_id_0001\n
app_id_00001\n
your nonce string\n
1702377418\n
mch_rsa_serial\n
857110231208020000000000049007\n
```

Second, the Raw data is the URL-encoded Base String. Example:

```
mch_id_0001%5Cn%0Aapp_id_00001%5Cn%0Ayour%20nonce%20string%5Cn%0A1702377418%5Cn%0Amch_rsa_serial%5Cn%0A857110231208020000000000049007%5Cn
```

Third, sign the Base String as follows:

```
class Demo {
    String sign(String requestData, String key) throws Exception {
        byte[] byteKey = Base64.decodeBase64(key);
        PKCS8EncodedKeySpec pkcs8EncodedKeySpec = new PKCS8EncodedKeySpec(byteKey);
        KeyFactory keyFactory = KeyFactory.getInstance("RSA");
        PrivateKey privateKey = keyFactory.generatePrivate(pkcs8EncodedKeySpec);
        Signature signature = Signature.getInstance("SHA256withRSA");
        signature.initSign(privateKey);
        signature.update(requestData.getBytes());
        return Base64.encodeBase64String(signature.sign());
    }
}
```

Finally, return the encoded data, payment signature, and signature type. Example:

```
{
  "rawData":
"mch_id_0001%5Cn%0Aapp_id_00001%5Cn%0Ayour%20nonce%20string%5Cn%0A1702377418%5Cn%0Amch_rsa_serial%5Cn%0A857110231208020000000000049007%5Cn",
  "paySign":
"PwUC/AmVaTfdBH+84d5XyX7Kge3pELyUcKiKhTPWLxTkHfaS9j2XEeMWRsd//1hjGQeDtuJ4uZTu2T0PP5hK/pHNi3r0suE8sUd3nyO/VeC0bTZsHQCfGciATN9P6q62abiZrN8d2eBlwPXeB2UOeCahLb11816ZFq3UjFA0pbzbISIDQ4r4GKgGCPO67aEpGYWhw7NX1r69MOwnBVTlczA76zL5kEioUfpu8i3MEgsmiAYg8mvSQy45TUfCxu9aNaK2VRcngoQAr31QOpvmSC6xbBXthb+iUPbgBW7r6vx9KOOU+ivVo1CfqfarmC2YR1tFW6uFXkWAlEEUqKw4qQ==",
  "signType": "SHA256withRSA"
}
```

2.7 Step 18: Pulling up the cashier

Step Description: After obtaining `rawData`, `paySign`, and `signType`, you need to invoke the payment API via JS Api to pull up Appleseed payment cashier.

> **Notes:** The window.payment object is not valid in other browsers.

```
const PayOrderRequest = {
rawData:'mch_id_0001%5Cn%0Aapp_id_00001%5Cn%0Ayour%20nonce%20string%5Cn%0A1702377418%5Cn%0Amch_rsa_serial%5Cn%0A857110231208020000000000049007%5Cn',
paySign:'PwUC/AmVaTfdBH+84d5XyX7Kge3pELyUcKiKhTPWLxTkHfaS9j2XEeMWRsd//1hjGQeDtuJ4uZTu2T0PP5hK/pHNi3r0suE8sUd3nyO/VeC0bTZsHQCfGciATN9P6q62abiZrN8d2eBlwPXeB2UOeCahLb11816ZFq3UjFA0pbzbISIDQ4r4GKgGCPO67aEpGYWhw7NX1r69MOwnBVTlczA76zL5kEioUfpu8i3MEgsmiAYg8mvSQy45TUfCxu9aNaK2VRcngoQAr31QOpvmSC6xbBXthb+iUPbgBW7r6vx9KOOU+ivVo1CfqfarmC2YR1tFW6uFXkWAlEEUqKw4qQ==',
signType: 'SHA256withRSA'
}

window.payment
.payOrder(PayOrderRequest)
.then(res => {
  /// success callback
  console.log('success', res)
})
.catch(error => {
  /// failure callback
  console.error('fail', error)
})
```

2.8 Step 23 to 25: Payment Results Notify

Step Description: After the user completes the payment, Appleseed will send a payment result notification to the `notifyUrl` provided during merchant registration. The merchant must acknowledge receipt of the notice and respond accordingly.

> **Notes:**
>
> - If Appleseed receives responses from merchants that HTTP status code is 200 or over time, Appleseed considers that notification fail, and Appleseed will again send the notification, try to improve the success rate of notifications.

Verify Signature

- Encryption does not ensure that the notification request come from Appleseed. Appleseed will sign the notification sent to the merchant and put the signature value in the HTTP header Signature of the notification. Merchants should verify the signature to confirm that the request come from Appleseed, and not any other third parties.

About the details of verifying signature algorithm, please refer to 《Verify Signature》.

Parameter Decrypt:

- The payment result notification is sent to the merchant's notification URL using the POST method. The notification includes encrypted payment result details.

Below is detail of the encrypted payment result decryption process:

1. Get the application's key introduced while merchant register, and record it as "key".

2. Get the algorithm described in `resource.algorithm` (currently AEAD_AES_256_GCM), `resource.nonce` and `resource.associated_data`.
3. Decrypt `resource.ciphertext` with "key", "nonce" and "associated_data" to Get a resource object in JSON form

Below is the sample code for encryption .

```
String decrypt(String associatedData, String nonce, String ciphertext) throws Exception
{
    Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
    cipher.init(
            2,
            new SecretKeySpec("your app secret key", "AES"),
            new GCMParameterSpec(128, nonce.getBytes(StandardCharsets.UTF_8)));
    if (associatedData != null) {
        cipher.updateAAD(associatedData.getBytes(StandardCharsets.UTF_8));
    }
    return new String(cipher.doFinal(Base64.getDecoder().decode(ciphertext)),
StandardCharsets.UTF_8);
}
```

Refer to the **OpenAPI** document for more request, response details, and error codes.

二、API Authentication

1 RSA

How to generate a request signature

The merchant can generate the requested signature by following the steps below. At the end of this section, we have prepared demonstration code in the Java programming language for developers to refer to.

The payment API requires the merchant to sign the request, and the payment will verify the signature after receiving the
request. If the signature verify fails, the payment API will refuse to process the request.

1.1 Preparation

The merchant needs to have a payment merchant number and a private key necessary for signature.

1.2 Construct signature string

We hope that the merchant's technical developers will construct the signature string according to the rules agreed in
the current document. Payments will construct signature strings in the same way. If the merchant constructs the
signature string in the wrong way, the signature verification will fail. The specific format of the signature string

will be explained below.

There are five lines in the signature string, each line has a parameter. Ends with \n (newline character, ASCII encoding
value 0x0A), including the last line. If the parameter itself ends with \n, a \n also needs to be appended.

```
HTTP request method\n
URL\n
Request timestamp\n
Request a random string\n
Request message body\n
```

We call interface of "pre order" on the command line, introducing developers step by step on how to perform request signing. According to the interface document, the URL to pre order is https://todo/v1/pay/pre-transaction/order/place, the request method is POST.

The first step is to get the HTTP request method (GET, POST), etc.

```
POST
```

The second step is to get the absolute URL of the request and remove the domain name part to get the URL participating in the signature.

```
/v1/pay/pre-transaction/order/place
```

The third step is to get the current timestamp of the system when the request is initiated, that is, 00:00:00 on January 1, 1970 Greenwich Time (08:00:00 on January 1, 1970, Beijing time)

The total number of seconds since now as the request timestamp. Payment will refuse to process requests initiated a long
time ago. Merchants are advised to keep the time of their own systems accurate.

```
$ date +%s
1554208460
```

The fourth step is to generate a request random string. We recommend the random number generation algorithm as follows:
call the random number function to generate and convert the obtained value into a string. Here, we generate one directly using the command line.

> hexdump is installed by default on most systems. If hexdump is not installed on your system, you can install it with
> the following command:

```
# Ubuntu/Debian System
sudo apt-get install hexdump
# CentOS/Fedora/RHEL System
sudo yum install hexdump
# Arch Linux System
sudo pacman -S hexdump
```

```
$ hexdump -n 16 -e '4/4 "%08X" 1 "\n"' /dev/random
PlggmuzaafHhqADY6Gg5YczBCJqFNVS1
```

The fifth step is to get the request body in the request.

- When the request method is GET, the message body is empty.
- When the request method is POST, please use the actual sent JSON message.

Step six: According to the aforementioned rules, the constructed request signature string is:

```
POST\n
/v1/pay/pre-transaction/order/place\n
1702377418\n
PlggmuzaafHhqADY6Gg5YczBCJqFNVS1\n
{"mchId": "Appleseed_toy_shop","appId": "Appleseed_toy_shop_pc_web","outBizId":
"20230102000100000010000023","timeExpire": 1723538571467,"description": "toy-
1.00","callbackInfo": "{}","amount": 100,"paymentProduct": "PCWeb","notifyUrl":
"https://...","redirectUrl": "https://..."}
\n
```

1.3 Calculate signature value

The signature functions provided by most programming languages support signing signature data. It is strongly
recommended that merchants call this type of function and use the merchant's private key to perform SHA256 with
RSA signature, and Base64 encoding the signature result to get the signature value.

```
$ echo -n -e \
"POST\n/v1/pay/pre-
transaction/order/place\n1702377418\nPlggmuzaafHhqADY6Gg5YczBCJqFNVS1\n{"mchId":
"Appleseed_toy_shop","appId": "Appleseed_toy_shop_pc_web","outBizId":
"20230102000100000010000023","timeExpire": 1723538571467,"description": "toy-
1.00ETB","callbackInfo": "{}","amount": 100,"paymentProduct": "PCWeb","notifyUrl":
"https://...","redirectUrl": "https://..."}\n" \
| openssl dgst -sha256 -sign apiclient_key.pem \
| openssl base64 -A
A0mN/eEgeK9Fw97PQ9NwNDduxVrj6EgrxlO2BLviZ1NRBZKt2CPGKB7Nsj/+UVQhDN+3XWqw3Ee9Tk0rowDn6GZ
meGFEjILClD1if3AfSaER3++K9v0HKl+tX9MbKI+i+6nuz2HpAEMndDwy8kib7CnSe6FOjRvH2++4McuOYkg=
```

1.4 Set HTTP header

The Payment Merchant API requires requests to pass signatures via the HTTP Authorization header. Authorization consists
of two parts: authorization type and signature information.

Below we use the command line to demonstrate how to generate a signature.

```
Authorization: <schema> <signature info>
```

The specific composition is:

1. schema, currently AES
2. signature info
    - mchId passed by the merchant making the request
    - Request a random string nonce_str
    - serial no of App Secret key
    - timestamp
    - signature value

> **Tip**
> Note: There is no order requirement for the above five signature information.

Below is an example of the Authorization header: (Note that the example may have line breaks due to formatting, and
the actual data should be on one line)

```
Authorization: SHA256withRSA mchid="your merchant
id",nonce_str="PlggmuzaafHhqADY6Gg5YczBCJqFNVS1",timestamp="1702377418",serial_no="your
merchant private key
serialNo",signature="A0mN/eEgeK9Fw97PQ9NwNDduxVrj6EgrxlO2BLviZ1NRBZKt2CPGKB7Nsj/+UVQhDN
+3XWqw3Ee9Tk0rowDn6GZmeGFEjILClD1if3AfSaER3++K9v0HKl+tX9MbKI+i+6nuz2HpAEMndDwy8kib7CnSe
6FOjRvH2++4McuOYkg="
```

Finally we can constitute an HTTP request that includes a signature.

> curl is installed by default on most systems. If curl is not installed on your system, you can install it with
> the following command:
>
> ```
> # Ubuntu/Debian System
> sudo apt-get install curl
> # CentOS/Fedora/RHEL System
> sudo yum install curl
> # Arch Linux System
> sudo pacman -S curl
> ```

```
$curl https://todo/v1/pay/pre-transaction/order/place -H 'Authorization: SHA256withRSA mchid="your merchant id",nonce_str="PlggmuzaafHhqADY6Gg5YczBCJqFNVS1",timestamp="1702377418",serial_no="your merchant private key serialNo",signature="A0mN/eEgeK9Fw97PQ9NwNDduxVrj6EgrxlO2BLviZ1NRBZKt2CPGKB7Nsj/+UVQhDN+3XWqw3Ee9Tk0rowDn6GZmeGFEjILClD1if3AfSaER3++K9v0HKl+tX9MbKI+i+6nuz2HpAEMndDwy8kib7CnSe6FOjRvH2++4McuOYkg=="'
```

1.5 Demo code

Developers can check the relevant chapters of the SDK to get the libraries for the corresponding languages.

Below is a sample code for calculating the signature:

```
import okhttp3.HttpUrl;

import java.security.KeyFactory;
import java.security.PrivateKey;
import java.security.Signature;
import java.security.spec.PKCS8EncodedKeySpec;
import java.util.Base64;

class demo {

    void example() throws Exception {
        // Authorization: <schema> <signatureInfo>
        // GET - getSignatureInfo("GET", httpurl, "")
        // POST - getSignatureInfo("POST", httpurl, json)
        String schema = "SHA256withRSA";
        HttpUrl httpurl = HttpUrl.parse("https://todo/v1/pay/pre-transaction/order/place");
        System.out.println(schema + " " + getSignatureInfo("POST", httpurl,
                ""{\"mchId\": \"Appleseed_toy_shop\",\"appId\": \"Appleseed_toy_shop_pc_web\",\"outBizId\": \"20230102000100000010000023\",\"timeExpire\": 1723538571467,\"description\": \"toy-1.00ETB\",\"callbackInfo\": \"{}\",\"amount\": 100,\"paymentProduct\": \"PCWeb\",\"notifyUrl\": \"https://...\",\"redirectUrl\": \"https://...\"}\n"));
    }

    String sign(String requestData, String key) throws Exception {
        byte[] byteKey = Base64.decodeBase64(key);
        PKCS8EncodedKeySpec pkcs8EncodedKeySpec = new PKCS8EncodedKeySpec(byteKey);
        KeyFactory keyFactory = KeyFactory.getInstance("RSA");
        PrivateKey privateKey = keyFactory.generatePrivate(pkcs8EncodedKeySpec);
        Signature sign = Signature.getInstance("SHA256withRSA");
        sign.initSign(privateKey);
        sign.update(requestData.getBytes());
        return Base64.encodeBase64String(sign.sign());
    }
```

```
    String getSignatureInfo(String method, HttpUrl url, String body) throws Exception {
        String nonceStr = NonceUtil.createNonce(NONCE_LENGTH);
        long timestamp = System.currentTimeMillis() / 1000;
        String message = buildMessage(method, url, timestamp, nonceStr, body);
        String signature = sign(message, "your merchant private key");
        return "mchid=\"" + "your merchant id" + "\","
                + "nonce_str=\"" + nonceStr + "\","
                + "timestamp=\"" + timestamp + "\","
                + "serial_no=\"" + "your merchant private key serialNo" + "\","
                + "signature=\"" + signature + "\"";
    }

    String buildMessage(String method, HttpUrl url, long timestamp, String nonceStr,
String body) {
        String canonicalUrl = url.encodedPath();
        if (url.encodedQuery() != null) {
            canonicalUrl += "?" + url.encodedQuery();
        }
        return method + "\n"
                + canonicalUrl + "\n"
                + timestamp + "\n"
                + nonceStr + "\n"
                + body + "\n";
    }
}
```

2 AES

How to generate a request signature

The merchant can generate the requested signature by following the steps below. At the end of this section, we have
prepared demo code in the Java programming language for developers to refer to.

The payment API requires the merchant to sign the request, and the payment will verify the signature after receiving the
request. If the signature verify fails, the payment API will refuse to process the request.

2.1 Preparation

The third party needs to register the application (identified by App Id) in advance and possess an application secret
key (App Secret Key)

2.2 Construct signature string

We hope that the merchant's technical developers will construct the signature string according to the rules agreed in
the current document. Payments will construct signature strings in the same way. If the merchant constructs the
signature string in the wrong way, the signature verification will fail. The specific format of the signature string
will be explained below.

There are five lines in the signature string, each line has a parameter. Ends with \n (newline character, ASCII encoding
value 0x0A), including the last line. If the parameter itself ends with \n, a \n also needs to be appended.

```
HTTP request method\n
URL\n
Request timestamp\n
Request a random string\n
Request message body\n
```

We call interface of "get openid" on the command line, introducing developers step by step on how to perform request signing. According to the interface document, the URL to get openid is https://todo/v1/pay/credential/openid, the request method is POST.

The first step is to get the HTTP request method (GET, POST), etc.

```
POST
```

The second step is to get the absolute URL of the request and remove the domain name part to get the URL participating in the signature.

```
/v1/pay/credential/openid
```

The third step is to get the current timestamp of the system when the request is initiated, that is, 00:00:00 on January 1, 1970 Greenwich Time (08:00:00 on January 1, 1970, Beijing time)

The total number of seconds since now as the request timestamp. Payment will refuse to process requests initiated a long
time ago. Merchants are advised to keep the time of their own systems accurate.

```
$ date +%s
1702373823
```

The fourth step is to generate a request random string. We recommend the random number generation algorithm as follows:
call the random number function to generate and convert the obtained value into a string. Here, we generate one directly

using the command line.

> hexdump is installed by default on most systems. If hexdump is not installed on your system, you can install it with
> the following command:
>
> ```
> # Ubuntu/Debian System
> sudo apt-get install hexdump
> # CentOS/Fedora/RHEL System
> sudo yum install hexdump
> # Arch Linux System
> sudo pacman -S hexdump
> ```

```
$ hexdump -n 16 -e '4/4 "%08X" 1 "\n"' /dev/random
z0d1twz0henQWNwzQDRRFuuemZgCb9nS
```

The fifth step is to get the request body in the request.

- When the request method is GET, the message body is empty.
- When the request method is POST, please use the actual sent JSON message.

Step 6: According to the aforementioned rules, the constructed request signature string is:

```
POST\n
/v1/pay/credential/openid\n
1702373823\n
z0d1twz0henQWNwzQDRRFuuemZgCb9nS\n
{"token":"4cf7bce965fc3b5d8eccc479f35e276b3b7a8ba027a3fbd9a59ad41fc64bc8f3"}
\n
```

2.3 Calculate signature value

The signature functions provided by most programming languages support signing signature data. It is strongly
recommended that merchants call this type of function and use the App Secret
Key) performs Advanced Encryption Standard (AES) encryption on the signature string and uses Galois/Counter
mode (GCM) for integrity protection, using the encrypted value as the signature; and Base64 encoding the signature
result to get the signature value.

For the calculation logic, please refer to the `sign` method in the following `5. Demo code` example

2.4 Set HTTP header

The Payment Merchant API requires requests to pass signatures via the HTTP Authorization header. Authorization consists
of two parts: authorization type and signature information.

Below we use the command line to demonstrate how to generate a signature.

```
Authorization: <schema> <signature info>
```

The specific composition is:

1. schema, currently AES
2. signature info
    - AppId passed by the merchant making the request
    - Request a random string nonce_str
    - serial no of App Secret key
    - timestamp
    - signature value

> **Tip**
> Note: There is no order requirement for the above five signature information.

Below is an example of the Authorization header: (Note that the example may have line breaks due to formatting, and the actual data should be on one line)

```
Authorization: AES
appid="APPID_GIFT_CARD",serial_no="123",nonce_str="z0d1twz0henQWNwzQDRRFuuemZgCb9nS",ti
mestamp="1702373823",signature="/NMImxl/7+mrVjUbCvdBxMrhk5YolCV0vpzQ7aydXzWX9gYEKziznIE
WRZWeKfEatxz+TCAgcWPHGqt/e2EC0aCVTzVJ0jPMLqhfIr8sSGAuv9/OkZJOhoXXzrsxA51hqGCyELm0Q54zJo
KG9au+SjqkVBGi3PgaFu0v82xojXaDijDk0kVlhHZ5FuVxFB6HLI7jgwbTe6CVMZyy22AgDuzicRnoloky2mIp2
cxnqH3rkZDu3V/FfSgYmzs="
```

Finally we can constitute an HTTP request that includes a signature.

> curl is installed by default on most systems. If curl is not installed on your system, you can install it with
> the following command:
>
> ```
> # Ubuntu/Debian System
> sudo apt-get install curl
> # CentOS/Fedora/RHEL System
> sudo yum install curl
> # Arch Linux System
> sudo pacman -S curl
> ```

```
$curl https://todo/v1/credential/openid -H 'Authorization: AES
appid="APPID_GIFT_CARD",serial_no="123",nonce_str="z0d1twz0henQWNwzQDRRFuuemZgCb9nS",ti
mestamp="1702373823",signature="/NMImxl/7+mrVjUbCvdBxMrhk5YolCV0vpzQ7aydXzWX9gYEKziznIE
WRZWeKfEatxz+TCAgcWPHGqt/e2EC0aCVTzVJ0jPMLqhfIr8sSGAuv9/OkZJOhoXXzrsxA51hqGCyELm0Q54zJo
KG9au+SjqkVBGi3PgaFu0v82xojXaDijDk0kVlhHZ5FuVxFB6HLI7jgwbTe6CVMZyy22AgDuzicRnoloky2mIp2
cxnqH3rkZDu3V/FfSgYmzs="'
```

2.5 Demo code

Developers can check the relevant chapters of the SDK to get the libraries for the corresponding languages.

Below is a sample code for calculating the signature:

```
import okhttp3.HttpUrl;

import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;

class demo {

    void example() throws Exception {
        // Authorization: <schema> <signatureInfo>
        // GET - getSignatureInfo("GET", httpurl, "")
        // POST - getSignatureInfo("POST", httpurl, json)
        String schema = "AES";
        HttpUrl httpurl = HttpUrl.parse("https://todo/v1/credential/openid");
        System.out.println(schema + " " + getSignatureInfo("POST", httpurl,
                "
{\"token\":\"4cf7bce965fc3b5d8eccc479f35e276b3b7a8ba027a3fbd9a59ad41fc64bc8f3\"}"));
    }

    String sign(String content, String keyStr) throws Exception {
        SecretKey secretKey = new SecretKeySpec(Base64.getDecoder().decode(keyStr),
"AES");
        Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
        cipher.init(1, secretKey);
        byte[] encryptData = cipher.doFinal(content.getBytes());
        byte[] message = new byte[12 + content.getBytes().length + 16];
        System.arraycopy(cipher.getIV(), 0, message, 0, 12);
        System.arraycopy(encryptData, 0, message, 12, encryptData.length);
        return Base64.getEncoder().encodeToString(message);
    }

    String getSignatureInfo(String method, HttpUrl url, String body) throws Exception {
        String nonceStr = "your nonce string";
        long timestamp = System.currentTimeMillis() / 1000;
        String message = buildMessage(method, url, timestamp, nonceStr, body);
```

```
        String signature = sign(message, "your app secret key");
        return "appid=\"" + "your app id" + "\","
                + "nonce_str=\"" + nonceStr + "\","
                + "timestamp=\"" + timestamp + "\","
                + "serial_no=\"" + "your app secret key serialNo" + "\","
                + "signature=\"" + signature + "\"";
    }

    String buildMessage(String method, HttpUrl url, long timestamp, String nonceStr,
String body) {
        String canonicalUrl = url.encodedPath();
        if (url.encodedQuery() != null) {
            canonicalUrl += "?" + url.encodedQuery();
        }
        return method + "\n"
                + canonicalUrl + "\n"
                + timestamp + "\n"
                + nonceStr + "\n"
                + body + "\n";
    }
}
```

3 Verify Signature

Merchants can follow the steps below to verify the signature of the response or callback.

If the merchant's request signature is correctly verified, Appleseed will include the response signature in the response HTTP header. We suggest that merchants verify the response signature.
Similarly, Appleseed will include the signature of the callback message in the HTTP header of the callback. Merchants
must verify the signature of the callback to ensure that the callback is sended by Appleseed.

3.1 Construct A Signature String

First, the merchant needs to get the following information from the response.

- timestamp in the HTTP header Timestamp
- random string in the HTTP header Nonce
- response body

Second, construct the response signature string according to the following rules. The signature string include three
lines, each line should end with \n. \n is a line break character (ASCII code value is 0x0A).

```
timestamp\n
random string\n
response body\n
```

Below is an example of getting openid response:

```
HTTP/1.1 200 OK
Date: Fri, 15 Dec 2023 05:45:06 GMT
Content-Type: application/json
Nonce: HLOaFrFKIJKP070k8G4wQQHqziYccBvI
Signature:
vMWFZ4XzBvDnr/TWmKJnlIJ/zYvu7ex7Enpmu5j0HLwMTY95Be2YPvXeYwtGWBXnogDmx0Hsg6E02A66DgqbYHc
+OT40E/MP2iWN5gXMZzqfZjWGS5gXYpPCuARYjrBVGwTep411dIfaNnKeFgPZjkQYY26zlsp+0xy+
Timestamp: 1702619106
Serial: 123
{"token" : "4cf7bce965fc3b5d8eccc479f35e276b3b7a8ba027a3fbd9a59ad41fc64bc8f3"}
```

The verify signature string should be:

```
1702619106
HLOaFrFKIJKP070k8G4wQQHqziYccBvI
{"token" : "4cf7bce965fc3b5d8eccc479f35e276b3b7a8ba027a3fbd9a59ad41fc64bc8f3"}
```

3.2 Get the Response Signature

The response signature of Appleseed is located in the HTTP header Signature.

```
Signature:
vMWFZ4XzBvDnr/TWmKJnlIJ/zYvu7ex7Enpmu5j0HLwMTY95Be2YPvXeYwtGWBXnogDmx0Hsg6E02A66DgqbYHc
+OT40E/MP2iWN5gXMZzqfZjWGS5gXYpPCuARYjrBVGwTep411dIfaNnKeFgPZjkQYY26zlsp+0xy+
```

Then, use Base64 to decode the response signature.

3.3 Verify Signature

Below is a sample code for `SHA256withRSA` algorithm verify signature:

```
import java.security.KeyFactory;
import java.security.PublicKey;
import java.security.Signature;
import java.security.spec.X509EncodedKeySpec;
import java.util.Base64;

class SHA256withRSA_Demo {
    boolean verify(String message, String signature) throws Exception {
        byte[] byteKey = Base64.getDecoder().decode("Appleseed rsa public key");
        X509EncodedKeySpec x509EncodedKeySpec = new X509EncodedKeySpec(byteKey);
        KeyFactory keyFactory = KeyFactory.getInstance("RSA");
        PublicKey publicKey = keyFactory.generatePublic(x509EncodedKeySpec);

        Signature verifySign = Signature.getInstance("SHA256withRSA");
        verifySign.initVerify(publicKey);
        verifySign.update(message.getBytes());
```

```
        return verifySign.verify(Base64.getDecoder().decode(signature));
    }
}
```

Below is a sample code for `AES` algorithm verify signature:

```
import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.spec.GCMParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;

class AES_Demo {

    boolean verify(String message, String signature) throws Exception {
        Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
        SecretKey key = new SecretKeySpec(Base64.getDecoder().decode("your app secret
key"), "AES");

        byte[] decodeBase64 = Base64.getDecoder().decode(signature);
        GCMParameterSpec params = new GCMParameterSpec(128, decodeBase64, 0, 12);
        cipher.init(2, key, params);
        byte[] decryptData = cipher.doFinal(decodeBase64, 12, decodeBase64.length -
12);

        return message == new String(decryptData);
    }
}
```

三、API List

1 Get OpenId

> BASIC

Path: /v1/pay/credential/openid

Method: POST

> REQUEST

Headers:

name	value	required	desc
Content-Type	application/json	YES	
Authorization	AES Reference	YES	

Request Body:

name	type	desc	required	constraint
token	String	Authorization token of this application, get it from the url parameter of the app that jump to the merchant's H5 page. the url such as: https://xxx.com/index.html?token=xxx	YES	1-64

Request Demo:

```
{
  "token": "b5bb89946b16e63e60d0ba7da9b5d63e1a476d45d774916cdb2f6be94c932803"
}
```

> RESPONSE

Headers:

name	value	required	desc
content-type	application/json;charset=UTF-8	YES	
Authorization	AES Reference	YES	

Body:

name	type	desc	required	constraint
openId	string	id specifying a user in the application	YES	1-32

Response Demo:

```
{
  "openId": "03ac9dd1580d2867001b6ddb05d0de8f"
}
```

Bussiness Error Code

error code	description
MERCHANT_APPLICATION_NOT_MATCHED	merchant application not matched
APPID_NOT_MATCH	appId not match
OPENID_INVALID	openId invalid
USER_NOT_EXIST	user not exist
TOKEN_NOT_EXIST	token not exist
TOKEN_ALREADY_EXPIRE	token already expire
TOKEN_CHECK_ERROR	token check fail
APPLICATION_NOT_EXIST	application not exist
MERCHANT_NOT_EXIST	merchant not exist

2 Get User Information

> BASIC

Path: /v1/pay/credential/user/info

Method: POST

> REQUEST

Headers:

name	value	required	desc
Content-Type	application/json	YES	
Authorization	AES Reference	YES	

Request Body:

name	type	desc	required	constraint
openId	string	id specifying a user in the application	YES	1-32
authToken	string	token granted for current operation, get it from the client-side JS SDK(SDK function is 'getAuthToken')	YES	1-64

Request Demo:

```
{
  "openId": "03ac9dd1580d2867001b6ddb05d0de8f",
  "authToken": "b5bb89946b16e63e60d0ba7da9b5d63e1a476d45d774916cdb2f6be94c932803"
}
```

> RESPONSE

Headers:

name	value	required	desc
content-type	application/json;charset=UTF-8	YES	
Authorization	AES Reference	YES	

Body:

name	type	desc	required	constraint
openId	string	id specifying a user in the application	YES	1-32
msisdn	string	mobile phone number	YES	1-32

Response Demo:

```
{
  "openId": "03ac9dd1580d2867001b6ddb05d0de8f",
  "msisdn": "+25196551200"
}
```

Bussiness Error Code:

error code	description
MERCHANT_APPLICATION_NOT_MATCHED	merchant application not matched
APPID_NOT_MATCH	appId not match
OPENID_INVALID	openId invalid
USER_NOT_EXIST	user not exist
TOKEN_NOT_EXIST	token not exist
TOKEN_NOT_BELONG_TO_OPENID	authToken don't belong to openId
TOKEN_ALREADY_EXPIRE	token already expire
TOKEN_CHECK_ERROR	token check fail
APPLICATION_NOT_EXIST	application not exist
MERCHANT_NOT_EXIST	merchant not exist

3 Order Placement

> BASIC

Path: /v1/pay/pre-transaction/order/place

Method: POST

> REQUEST

Headers:

name	value	required	desc
Content-Type	application/json	YES	
Authorization	RSA Reference	YES	

Request Body:

name	type	desc	required	constraint
mchId	string	merchant id, given when registered	YES	1-128
appId	string	application id, given when registered	YES	1-128
outBizId	string	merchant order number	YES	1-128
timeExpire	long	pre pay order time expire, the unit is milliseconds	YES	
description	string	order description	NO	0-256
callbackInfo	string	information that giving back to issuer when callback	NO	0-2048
amount	long	transaction amount	YES	> 0
currency	string	to specify currencies in request URI and body parameters, use three-character ISO-4217 codes. Note: This currency does not support decimals.	YES	3
paymentProduct	string	payment product `InAppH5` : in app h5 Payment	YES	1-32
notifyUrl	string	the url is used to notify the merchant of the payment result	NO	0-2048
redirectUrl	string	the url is used to redirect back to the merchan page	NO	0-2048

Request Demo:

```
{
  "mchId": "Appleseed_toy_shop",
  "appId": "Appleseed_toy_shop_pc_web",
  "outBizId": "20230102000100000010000023",
  "timeExpire": 1723538571467,
  "description": "toy-1.00",
  "callbackInfo": "{}",
  "amount": 100,
  "currency": 'USD',
  "paymentProduct": "InAppH5",
  "notifyUrl": "https://...",
  "redirectUrl": "https://..."
}
```

> RESPONSE

Headers:

name	value	required	desc
content-type	application/json;charset=UTF-8	YES	
Authorization	RSA Reference	YES	

Body:

name	type	desc	required	constraint
prepayId	string	prepay id	YES	1-128

Response Demo:

```
{
  "prepayId": "85711023120802000000000049007"
}
```

Bussiness Error Code:

error code	description
MERCHANT_APPLICATION_NOT_MATCHED	merchant application not matched
PAYMENT_SIGN_CHECK	payment sign check failed
PRE_PAY_ORDER_NULL	pre pay order null
PRE_PAY_ORDER_INFO_NOT_CONSTANT	pre pay order info not constant
PRE_PAY_ORDER_AMOUNT_CHECK_FAILURE	pre pay order amount check failure
PRE_PAY_ORDER_OCCUPIED	pre pay order is occupied
APPID_NOT_MATCH	appId not match
APPLICATION_NOT_EXIST	application not exist
MERCHANT_NOT_EXIST	merchant not exist
PAYMENT_APP_CONTRACT_NOT_EXIST	payment application not exist contract

4 Payment Result Query

> BASIC

Path: /v1/pay/transaction/result

Method: POST

> REQUEST

Headers:

name	value	required	desc
Content-Type	application/json	YES	
Authorization	RSA Reference	YES	

Request Body:

name	type	desc	required	constraint
outBizId	string	merchant order number	YES	1-128

Request Demo:

```
{
  "outBizId": "1234567890"
}
```

> RESPONSE

Headers:

name	value	required	desc
content-type	application/json;charset=UTF-8	YES	
Authorization	RSA Reference	YES	

Body:

name	type	desc	required	constraint
amount	long	transaction amount, expressed in the smallest monetary unit	YES	> 0
currency	string	to specify currencies in request URI and body parameters, use three-character ISO-4217 codes.	YES	1-3
orderId	string	Pay or Refund order number	YES	1-128
orderStatus	string	Pay or Refund order status `SUCCESS`: Pay or Refund success `PROCESSING`: Pay or Refund processing `CLOSED`: Closed `FAIL`: Pay or Refund failed	YES	1-16

Response Demo:

```
{
  "amount": "100",
  "currency": "USD",
  "orderId": "857112240108010000000000461000",
  "orderStatus": "PROCESSING"
}
```

Bussiness Error Code:

error code	description
PRE_PAY_ORDER_NULL	pre pay order null
PAY_ORDER_NULL	pay order null

5 Payment Notification

> BASIC

Method: POST

Callback URL: This url is provided during merchant registration, which requires an HTTPS address. Please ensure that
the callback URL is accessible externally and does not carry suffix parameters, otherwise the merchant may not be able
to receive Appleseed callback notification information.

> REQUEST

Headers:

name	value	required	desc
Content-Type	application/json	YES	
Authorization	RSA Reference	YES	

Request Body:

name	type	desc	required	constraint
serialNo	string	serial number of app secret key	YES	1-32
prepayId	string	prepay id	YES	1-32
algorithm	string	The algorithm that encrypts the result data. Only AEAD_AES_256_GCM is supported. Example: AEAD_AES_256_GCM	YES	1-32
associatedData	string	Additional data	NO	1-16
nonce	string	The random string used for encryption	YES	1-32
ciphertext	string	The ciphertext of the refund result encoded with Base64	YES	1-1048576

Request Demo:

```
{
  "serialNo": "1",
  "prepayId": "85711023120802000000000049007",
  "algorithm": "AEAD_AES_256_GCM",
  "originalType": "transaction",
  "associatedData": "",
  "nonce": "...",
  "ciphertext": "..."
}
```

▶ ciphertext decrypted field

name	type	desc	required	constraint
appId	string	application id, given when registered	YES	1-128
mchId	string	merchant id, given when registered, given when registered	YES	1-128
outBizId	string	merchant order number	YES	1-128
prepayId	string	prepay id	YES	1-128
paymentOrderId	string	payment order id	YES	1-128
tradeType	string	trade type, only support Payment and Refund	YES	1-32
status	string	trade status, only support SUCCESS	YES	1-32
callbackInfo	string	callback information that giving when place order	YES	0-2048
finishTime	long	payment or refund completion time	YES	1-128
orderAmount	long	order amount	YES	> 0
paidAmount	long	paid amount	YES	> 0
currency	string	to specify currencies in request URI and body parameters, use three-character ISO-4217 codes. Note: This currency does not support decimals.	YES	3
originalOutBizId	string	out biz id of original order	NO	1-128
originalPrepayId	string	prepay id of original order	NO	1-128
originalPaymentOrderId	string	payment order id of original order	NO	1-128
originalOrderAmount	long	order amount of original order	NO	> 0
originalPaidAmount	long	paid amount of original order	NO	> 0
paymentProduct	string	payment product	YES	1-32
description	string	description	YES	0-2048

RESPONSE

Headers:

name	value	required	desc
content-type	application/json;charset=UTF-8	YES	
Authorization	RSA Reference	YES	

Body:

name	type	desc	required	constraint
code	string	For more information, see the returned status code description below. Example: SUCCESS	YES	1-32
message	string	Returned message, which presents the cause of the error. Example: System error	YES	1-256

Response Demo:

```
{
  "code": "SUCCESS"
}
```

6 Common Error Code

error code	description
SYSTEM_UNSTABLE	system is unstable
PARAM_ILLEGAL	params is illegal
PARAM_IS_NULL	params is null
REQUEST_METHOD_NOT_SUPPORT	request method not support
REPEAT_REQUEST	repeat request, please try again later
ALGORITHM_TYPE_NOT_SUPPORT	algorithm type not support
SIGNATURE_VERIFY_FAILED	signature verification failed
ALGORITHMTYPE_NOT_EXPECT	algorithm type not expect
RSA_KEY_SERIAL_NO_NOT_MATCH	rsa key serial no not match