



PHP SDK Developers Guide

Table of Contents

- 1. About this document3
- 2. Integrations steps3
 - a) Install PHP SDK Package.....3
 - b) Merchant configuration.....3
 - c) Payment data configuration4
- 3. Payment flow6
- 4. Maintenance Operations7
 - a) Capture7
 - b) Refund.....7
 - c) Void8
 - d) Check Status.....8
 - e) Installments Plans9
- 5. Integration Channels.....10
 - a) Redirect.....10
 - b) Redirect Installments.....11
 - c) Standard Checkout11
 - d) Standard Checkout Installments.....12
 - e) Custom Checkout.....12
 - f) Custom Checkout Installments.....13
- 6. Apple Pay Integration14
 - a) Apple Pay Button14
 - b) Apple Pay Validation.....15
 - c) Apple Pay Commands15
- 7. Trusted Channels17
 - a) MOTO.....17
 - b) Recurring.....18
 - c) Trusted19
- 8. Response Handler20
- 9. 3DS Modal.....21
- 10. Webhook.....23
- 11. Error codes.....23

1. About this document

This document describes how to integrate the PHP SDK into your solution.

2. Integrations steps

a) Install PHP SDK Package

Install the PHP SDK Package from or of ? your solution with composer or download it from the GitHub repository and then run the composer update command in terminal to install all the dependencies.

b) Merchant configuration

As a merchant you need to send to the gateway some properties (Figure 1). These properties must be put into an array and set with the following method (Figure 2). If you want integration with Apple Pay all the properties that contains “Apple_” must be added, otherwise those properties are not required.

```
<?php
return [
    'merchant_identifier'    => '*****',
    'access_code'           => '*****',
    'SHARequestPhrase'      => '*****',
    'SHAResponsePhrase'     => '*****',
    'SHAType'               => '*****',
    'sandbox_mode'          => true,

    'Apple_AccessCode'      => '*****',
    'Apple_SHARequestPhrase' => '*****',
    'Apple_SHAResponsePhrase' => '*****',
    'Apple_SHAType'         => '*****',
    'Apple_DisplayName'     => 'Test Apple store',
    'Apple_DomainName'      => 'https://store.local.com',
    'Apple_SupportedNetworks' => ["visa", "masterCard", "amex", "mada"],
    'Apple_SupportedCountries' => [],
    'Apple_CertificatePath'  => '**path**to**certificate**',
    'Apple_CertificateKeyPath' => '**path**to**certificate**key**',
    'Apple_CertificateKeyPass' => 'apple*certificate*password',

    // folder must be created before
    'log_path'              => __DIR__ . '/tmp/aps.log',
    '3ds_modal'             => true,
    'debug_mode'            => false,
    'locale'                => 'en',
];
```

Figure 1 All the merchant configuration properties

```
// load merchant configuration
$merchantParams = include 'merchant_config.php';

// set merchant configuration one time
APSMerchant::setMerchantParams($merchantParams);
```

Figure 2 The method used for setting the merchant configuration

c) Payment data configuration

As a merchant you need to send to the gateway the payment details (Figure.3). These details must be put into an array and set within the “setPaymentData” method (Figure 4). The “merchant_reference” is the customer order number, in Figure 3 you can see an example of order number.

```
<?php
return [
    'merchant_reference'=> 'O-00001-'.rand(1000, 99999) ,
    'amount'             => 3197.00,
    'currency'           => 'AED',
    'language'           => 'en',
    'customer_email'     => 'test@aps.com',

    'order_description' => 'Test product 1',
];
```

Figure 3 All the payment details properties

In Figure 4 you can see how the credit card redirect payment method is used. Payment data is set with the payment details, then set the authorization/purchase command, set your callback URL and render the information needed for your client page.

```
<div>
  <?php
  try {
    echo (new CCRedirect())
      ->setPaymentData($paymentData)
      ->useAuthorizationCommand()
      ->setCallbackUrl('callback-url.php')
      ->render([
        'button_text' => 'Place order with Authorization'
      ]);
  } catch (APSEException $e) {
    echo 'SETUP ERROR: ' . $e->getMessage();
  }
  ?>
</div>
<div>
  <?php
  try{
    echo (new CCRedirect())
      ->setPaymentData($paymentData)
      ->usePurchaseCommand()
      ->setCallbackUrl('callback-url.php')
      ->render([
        'button_text' => 'Place order with Purchase'
      ]);
  } catch (APSEException $e) {
    echo 'SETUP ERROR: ' . $e->getMessage();
  }
  ?>
</div>
```

Figure 4 Credit card redirect payment method

3. Payment flow

Below we can see a high-level diagram with the important systems which are involved in the payment flow:

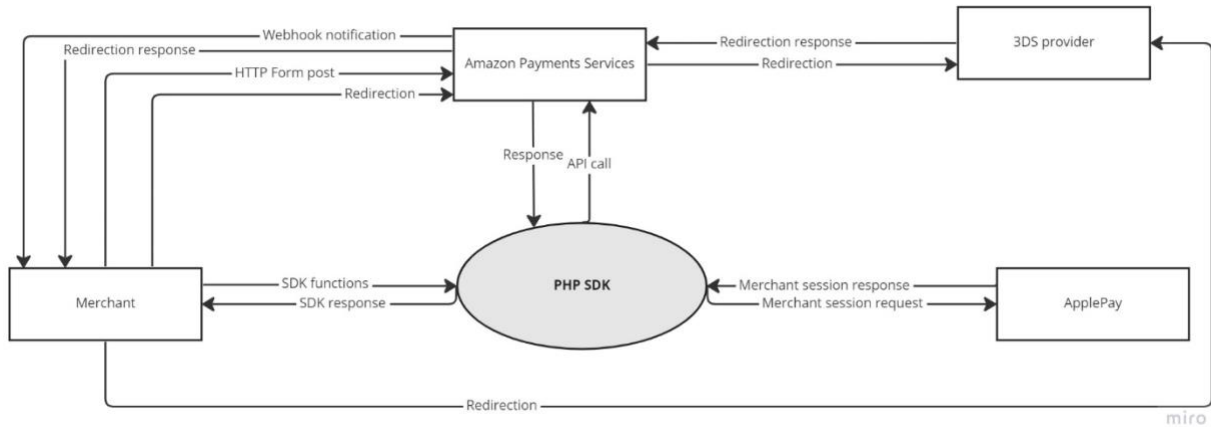


Figure 5 Payment flow diagram

In the above picture the Merchant will use the SDK in the following situation:

- Get the HTML Forms necessary to initiate one payment integration (Redirect, Standard iframe checkout or Custom checkout)
- Make an API call to APS through the APS SDK for completing the payment process (Authorization/Purchase) and maintenance operations (Capture/Void/Refund)
- Validate the webhook notification and signature sent by the payment gateway

4. Maintenance Operations

All maintenance operations provided by the PHP SDK can be found on “/maintenance.php” tab.

a) Capture

An operation that allows the Merchant to capture the authorized amount of a payment. For more details, regarding parameters check the following link: [Capture Operation](#).

```
<div>
  <?php
  try {
    $paymentData = [
      'merchant_reference' => '1234567890',
      'amount'             => 3197.00,
      'currency'           => AED,
      'language'           => 'en',
    ];

    $callResult = (new PaymentCapture())->paymentCapture($paymentData);
    echo $callResult['response_message'];
  } catch (APSEException $e) {
    echo 'SETUP ERROR: ' . $e->getMessage();
  }
  ?>
</div>
```

Figure 6 Capture operation

b) Refund

An operation that returns the entire amount of a transaction, or returns part of it, after a successfully capture operation is done. For more details regarding parameters check the link: [Refund Operation](#).

```
<div>
  <?php
  try {
    $paymentData = [
      'merchant_reference' => '1234567890',
      'amount'             => 3197.00,
      'currency'           => AED,
      'language'           => 'en',
    ];

    $callResult = (new PaymentRefund())->paymentRefund($paymentData);
    echo $callResult['response_message'];
  } catch (APSEException $e) {
    echo 'SETUP ERROR: ' . $e->getMessage();
  }
  ?>
</div>
```

Figure 7 Refund operation

c) Void

An operation that allows you to cancel the authorized amount after you have sent a successful authorization request. For more details, [Void Operation](#).

```
<div>
  <?php
  try {
    $paymentData = [
      'merchant_reference' => '1234567890',
      'language'           => 'en',
    ];

    $callResult = (new PaymentVoidAuthorization())
      ->paymentVoid($paymentData);
    echo $callResult['response_message'];
  } catch (APSEException $e) {
    echo 'SETUP ERROR: ' . $e->getMessage();
  }
  ?>
</div>
```

Figure 8 Void operation

d) Check Status

In case you need to verify the status of a transaction in progress you can do it by using the Check Status method. For more details, [Check Status Operation](#).

```
<div>
  <?php
  try {
    $paymentData = [
      'merchant_reference' => '1234567890',
      'language'           => 'en',
    ];

    $callResult = (new PaymentCheckStatus())
      ->paymentCheckStatus($paymentData);
    echo $callResult['response_message'];
  } catch (APSEException $e) {
    echo 'SETUP ERROR: ' . $e->getMessage();
  }
  ?>
</div>
```

Figure 9 Check status operation

e) Installments Plans

In case you need to see the installments plans you can do it by using the Get Installments Plans method. For more details, [Get Installments Plans Operation](#).

```
<div>
  <?php
  try {
    $paymentData = [
      'amount'           => 3197.00,
      'currency'         => 'AED',
      'language'         => 'en',
    ];

    $callResult = (new InstallmentsPlans())
      ->getInstallmentsPlans($paymentData);

    echo "<pre>";
    print_r($callResult);
    echo "</pre>";
  } catch (APSEException $e) {
    echo 'SETUP ERROR: ' . $e->getMessage();
  }
  ?>
</div>
```

Figure 10 Get installments plans operation

5. Integration Channels

The method which returns the form post which needs to be added in the html page, is called “render”. This method is found in the “FrontEndAdapter” class, this class is inherited by all the payment options.

a) Redirect

The class for Redirect payment option is called “CCRedirect”. This class can be used for Authorization or for Purchase command. For example, see the code below (Figure. 11.).

```
<div>
  <?php
  try {
    echo (new CCRredirect())
      ->setPaymentData($paymentData)
      ->useAuthorizationCommand()
      ->setCallbackUrl('callback-url.php')
      ->render([
        'button_text' => 'Place order with Authorization'
      ]);
  } catch (APSEException $e) {
    echo 'SETUP ERROR: ' . $e->getMessage();
  }
  ?>
</div>
<div>
  <?php
  try {
    echo (new CCRredirect())
      ->setPaymentData($paymentData)
      ->usePurchaseCommand()
      ->setCallbackUrl('callback-url.php')
      ->render([
        'button_text' => 'Place order with Purchase'
      ]);
  } catch (APSEException $e) {
    echo 'SETUP ERROR: ' . $e->getMessage();
  }
  ?>
</div>
```

Figure 11 Render Html form post for redirect payment option

b) Redirect Installments

The class for Redirect Installments payment option is called “InstallmentsCCRedirect”. This class can be used for Purchase command. For example, see the code below (Figure 12).

```
<div>
  <?php
  try {
    echo (new InstallmentsCCRedirect())
      ->setPaymentData($paymentData)
      ->setCallbackUrl('callback-url.php')
      ->render();
  } catch (APSEException $e) {
    echo 'SETUP ERROR: ' . $e->getMessage();
  }
  ?>
</div>
```

Figure 12 Render Html form post for redirect installments payment option

c) Standard Checkout

The class for Standard Checkout payment option is called “CCStandard”. This class can be used for Authorization or for Purchase command. For example, see the code below (Figure 13).

```
<div>
  <?php
  try {
    echo (new CCStandard())
      ->setPaymentData($paymentData)
      ->useAuthorizationCommand()
      ->setCallbackUrl('callback-url.php')
      ->render([
        'button_text' => 'Place order with Authorization'
      ]);
  } catch (APSEException $e) {
    echo 'SETUP ERROR: ' . $e->getMessage();
  }
  ?>
</div>
<div>
  <?php
  try {
    echo (new CCStandard())
      ->setPaymentData($paymentData)
      ->usePurchaseCommand()
      ->setCallbackUrl('callback-url.php')
      ->render([
        'button_text' => 'Place order with Purchase'
      ]);
  } catch (APSEException $e) {
    echo 'SETUP ERROR: ' . $e->getMessage();
  }
  ?>
</div>
```

Figure 13 Render Html form post for standard checkout payment option

d) Standard Checkout Installments

The class for Standard Installments payment option is called “InstallmentsCCStandard”. This class can be used for Purchase command. For example, see the code below (Figure 14).

```
<div>
  <?php
  try {
    echo (new InstallmentsCCStandard())
      ->setPaymentData($paymentData)
      ->setCallbackUrl('callback-url.php')
      ->render();
  } catch (APSEException $e) {
    echo 'SETUP ERROR: ' . $e->getMessage();
  }
  ?>
</div>
```

Figure 14 Render Html form post for standard installments payment option

e) Custom Checkout

The class for Custom Checkout payment option is called “CCCustom”. This class can be used for Authorization or for Purchase command. For example, see the code below (Figure 15).

```
<div>
  <?php
  try {
    echo (new CCCustom())
      ->setPaymentData($paymentData)
      ->useAuthorizationCommand()
      ->setCallbackUrl('callback-url.php')
      ->render([
        'button_text' => 'Place order with Authorization'
      ]);
  } catch (APSEException $e) {
    echo 'SETUP ERROR: ' . $e->getMessage();
  }
  ?>
</div>
<div>
  <?php
  try {
    echo (new CCCustom())
      ->setPaymentData($paymentData)
      ->usePurchaseCommand()
      ->setCallbackUrl('callback-url.php')
      ->render([
        'button_text' => 'Place order with Purchase'
      ]);
  } catch (APSEException $e) {
    echo 'SETUP ERROR: ' . $e->getMessage();
  }
  ?>
</div>
```

Figure 15 Render Html form post for custom checkout payment option

f) Custom Checkout Installments

First step to work with Custom Installments payment option you need to get the installments plans with the “getInstallmentsPlans” method which is defined in “InstallmentsPlans” class. See [Maintenance Operation Installments Plans](#) section for more details.

The second step to work with Custom Installments payment option is to handle the installments plans that the customer will select from the interface and get the selected plan code and issuer code to set them in session in order to be used at purchase.

The class for Custom Installments payment option is called “InstallmentsCCCustom”. This class can be used for Purchase command. For example, see the code below (Figure 16).

```
<div>
  <?php
  try {
    $installmentsPlansList = (new InstallmentsPlans())
      ->getInstallmentsPlans($paymentData);

    // save the plan code that will be used at authorization
    session_start();
    $_SESSION['plan_code'] = 'selected_plan_code';
    $_SESSION['issuer_code'] = 'selected_issuer_code';
    session_commit();

    echo (new InstallmentsCCCustom())
      ->setPaymentData($paymentData)
      ->setCallbackUrl('callback-url.php')
      ->render(
        [
          'installment_plans_list' => $installmentsPlansList,
        ]
      );
  } catch (APSEException $e) {
    echo 'SETUP ERROR: ' . $e->getMessage();
  }
  ?>
</div>
```

Figure 16 Render Html form post for custom installments payment option

6. Apple Pay Integration

a) Apple Pay Button

The class for Apple Pay payment option is called “ApplePayButton”. This class can be used for Authorization or for Purchase command. First you need to set extra parameters for payment data, see Figure 17.

```
$paymentData = [
    'merchant_reference'=> '1234567890',
    'amount'             => 1500.00,
    'currency'           => 'AED',
    'language'           => 'en',
    'customer_email'     => 'test@aps.com',
    'order_description' => 'Test product 1',
];

// ADDITIONAL Apple Pay required parameters
$paymentData['subtotal'] = 1245.00;
$paymentData['discount'] = 200;
$paymentData['shipping'] = 50;
$paymentData['tax']      = 5;
$paymentData['country']  = 'AE';
```

Figure 17 Apple Pay extra parameters

Now you can call the Apple Pay class to create the button for payment. For example, see the code below (Figure 18). [See merchant configuration for Apple Pay here.](#)

```
<div>
  <?php
  try {
    echo (new ApplePayButton())
      ->setPaymentData($paymentData)
      // apple specific settings
      ->setDisplayName($merchantParams['Apple_DisplayName'])
      ->setCurrencyCode($paymentData['currency'])
      ->setCountryCode($paymentData['country'])
      ->setSupportedCountries(
        $merchantParams['Apple_SupportedCountries']
      )
      ->setSupportedNetworks(
        $merchantParams['Apple_SupportedNetworks']
      )
      ->setValidationCallbackUrl('apple-pay-validation-callback.php')
      ->setCommandCallbackUrl(
        'apple-pay-authorization-or-purchase-callback.php'
      )
      ->render();
  } catch (APSEException $e) {
    echo 'SETUP ERROR: ' . $e->getMessage();
  }
  ?>
</div>
```

Figure 18 Apple Pay button usage

b) Apple Pay Validation

After the payment is initialized, you need to validate the session. You can validate the session by setting the validation callback that will trigger your validation handler.

```
// get the apple url that Safari sends us
$appleUrl = $_POST['url'] ?? '';

// the response must be JSON
header('Content-Type: application/json; charset=utf-8');

try {
    echo (new PaymentApplePaySession())->applePayValidateSession($appleUrl);
} catch (APSEException $e) {
    http_response_code(400);
    echo json_encode([
        'status' => 'fail',
        'message' => 'Session validation failed' . $e->getMessage(),
    ]);
}
exit;
```

Figure 19 Apple Pay validation

c) Apple Pay Commands

After validation you now can make an authorization or a purchase command. You can do that by setting the command callback that will trigger your command handler.

See Figure 20 and Figure 21 for more details.

```
header('Content-Type: application/json; charset=utf-8');

try {
    // collect data sent over by Safari
    // and prepare parameters to be sent to APS
    // retrieve and validate response from APS
    $callResult = (new PaymentApplePayAps())
        ->applePayAuthorization($paymentData);

    echo json_encode([
        'status' => 'success',
        'message' => $callResult['response_message'] ?? 'no message',
    ]);
} catch (APSEException $e) {
    http_response_code(400);

    echo json_encode([
        'status' => 'fail',
        'message' => 'APS purchase call failed' . $e->getMessage(),
    ]);
}
exit;
```

Figure 20 Apple Pay authorization

```
header('Content-Type: application/json; charset=utf-8');

try {
    // collect data sent over by Safari
    // and prepare parameters to be sent to APS
    // retrieve and validate response from APS
    $callResult = (new PaymentApplePayAps())
        ->applePayPurchase($paymentData);

    echo json_encode([
        'status' => 'success',
        'message' => $callResult['response_message'] ?? 'no message',
    ]);
} catch (APSEException $e) {
    http_response_code(400);

    echo json_encode([
        'status' => 'fail',
        'message' => 'APS purchase call failed' . $e->getMessage(),
    ]);
}
exit;
```

Figure 21 Apple Pay purchase

7. Trusted Channels

a) MOTO

MOTO channel enables you to process a range of transactions that do not follow the standard online shopping workflow. For example, your customer may want to pay you offline. By sending an order in the post, by calling you, or indeed in a face-to-face transaction. You can process offline transactions using the MOTO channel.

Note that the MOTO (Mobile Order/ Telephone Order) channel allows you to process MOTO transactions through the Amazon Payment Services API only if you have already established a token for your customer's payment card.

In Figure 22, you can see how to implement the recurring operation. You need to know the token name of the transaction and to set it to the payment data object. If the token name is not known then the credit card details like card number, expiry date and security code are required to be set to the payment data object.

For more details, [Moto Operation](#).

```
<div>
  <?php
  try {
    $paymentData = [
      'merchant_reference'=> '1234567890',
      'amount'              => 1500.00,
      'currency'            => 'AED',
      'language'            => 'en',
      'customer_email'      => 'test@aps.com',
      'order_description'   => 'Test product 1',
      'token_name'          => 'token-name',
      'customer_ip'         => '127.0.0.1'
    ];

    $callResult = (new PaymentMoto())->paymentMoto($paymentData);

    echo $callResult['response_message'];
  } catch (APSEException $e) {
    echo 'SETUP ERROR: ' . $e->getMessage();
  }
  ?>
</div>
```

Figure 22 Moto operation

b) Recurring

You can effortlessly configure secure, recurring payments for any defined billing cycle – whether daily, weekly, monthly, or annual. You do so through a workflow that is not much different from the normal checkout process.

In Figure 23, you can see how to implement the recurring operation. You need to know the token name of the transaction and to set it to the payment data object. If the token name is not known then the credit card details like card number, expiry date and security code are required to be set to the payment data object.

For more details, [Recurring Operation](#).

```
<div>
  <?php
  try {
    $paymentData = [
      'merchant_reference'=> '1234567890',
      'amount'              => 1500.00,
      'currency'            => 'AED',
      'language'            => 'en',
      'customer_email'      => 'test@aps.com',
      'order_description'   => 'Test product 1',
      'token_name'          => 'token-name',
    ];

    $callResult = (new PaymentRecurring())->paymentRecurring($paymentData);

    echo $callResult['response_message'];
  } catch (APSEException $e) {
    echo 'SETUP ERROR: ' . $e->getMessage();
  }
  ?>
</div>
```

Figure 23 Recurring operation

c) Trusted

If you are a PCI-certified merchant, you can collect your customers' credit card details on your checkout page and store the sensitive payment card data on your server. [Read more about PCI compliance here.](#)

PCI-compliant merchants can use the Amazon Payment Services trusted channel to submit payment card details so that Amazon Payment Services can execute transactions using the payment card details or token name. In the payment data object, you need to specify the "eci" property with the one of the following options: MOTO, RECURRING or ECOMMERCE.

```
try {
    $paymentData = [
        'merchant_reference'=> '1234567890',
        'amount'             => 1500.00,
        'currency'           => 'AED',
        'language'           => 'en',
        'customer_email'     => 'test@aps.com',
        'eci'                => 'ECOMMERCE',
        'customer_ip'        => '127.0.0.1',
        'card_number'        => '1234567890123456',
        'expiry_date'        => '1234',
        'card_security_code' => '123',
        'token_name'         => 'token-name',
    ];

    $callResult = (new PaymentTrusted())->paymentTrusted($paymentData);
    if (is_string($callResult)) {
        // and it requests 3ds validation,
        // which is not accessible to the client
        echo $callResult;
    } else {
        echo "<pre>";
        print_r($callResult);
        echo "</pre>";
    }
} catch (APSEException $e) {
    echo 'CALL DATA ERROR: ' . $e->getMessage();
}
```

Figure 24 Trusted operation

8. Response Handler

To handle the response for payment methods we have implemented the “ResponseHandler” class which takes the initial payment data.

This class also has the following methods for a proper response handling:

- validate
- process
- onSuccess (has a callback to handle the success case)
- onError (has a callback to handle the error case)
- onHtml
- handleResponse
- getResult

```
try {
    (new ResponseHandler($paymentData))
        ->onSuccess(function(APSResponse $responseHandlerResult) {...})
        ->onError(function(APSResponse $responseHandlerResult) {...})
        ->onHtml(function(
            string $htmlContent,
            APSResponse $responseHandlerResult
        ) {...})
        ->handleResponse();
} catch (APSEException $e) {
    include '_header.php';
    echo 'APS RESPONSE ERROR: ' . $e->getMessage();
    include '_footer.php';
}
```

Figure 25 Response handler usage on redirect page

9. 3DS Modal

The PHP SDK provides you with the code for handling the 3ds secure part. After calling Authorization or Purchase command, you get the response from the gateway. In the response, you can get the 3DS Secure URL which needs to be used with the modal provided by the SDK by calling the method “render” found in “Secure3dsModal” class. In this case the method needs a parameter called “3ds_url” that will be set with the obtained URL.

```
return (new Secure3dsModal())->render([
    '3ds_url' => '3-ds-url-from-response',
]);
```

Figure 26 Secure 3ds modal usage

Implicitly Standard, Custom, Installments Standard and Installments Custom payment methods has 3ds. In the redirect page, we need to handle it with a redirect to your success and fail page.

In “onHtml” callback function we need to handle the Standard and Installments Standard payment methods because they use an iframe for credit card details, so after that we redirect them to the 3ds url.

```
try {
    (new ResponseHandler($paymentData))
        ->onSuccess(function(APSResponse $responseHandlerResult) {
            // the payment transaction was a success
            // do your thing and process the response

            // redirect user to the success page
            $redirectParams = $responseHandlerResult->getRedirectParams();
            header(
                'Location: order_success.php?' . $redirectParams
            );
            exit;
        })
        ->onError(function(APSResponse $responseHandlerResult) {...})
        ->onHtml(function(
            string $htmlContent,
            APSResponse $responseHandlerResult
        ) {
            if ($responseHandlerResult->isStandardImplementation()) {
                header('Location: ' . $responseHandlerResult->get3dsUrl());
                exit;
            } else {
                echo $htmlContent;
            }
        })
        ->handleResponse();
} catch (APSEException $e) {
    include '_header.php';
    echo 'APS RESPONSE ERROR: ' . $e->getMessage();
    include '_footer.php';
}
```

Figure 27 Payment method success case

```
try {
    (new ResponseHandler($paymentData))
        ->onSuccess(function(APSResponse $responseHandlerResult) {...})
        ->onError(function(APSResponse $responseHandlerResult) {
            // the payment failed
            // process the response

            // redirect user to the error page
            $redirectParams = $responseHandlerResult->getRedirectParams();
            header(
                'Location: order_failed.php?' . $redirectParams
            );
            exit;
        })
        ->onHtml(function(
            string $htmlContent,
            APSResponse $responseHandlerResult
        ) {
            if ($responseHandlerResult->isStandardImplementation()) {
                header('Location: ' . $responseHandlerResult->get3dsUrl());
                exit;
            } else {
                echo $htmlContent;
            }
        })
        ->handleResponse();
} catch (APSEException $e) {
    include '_header.php';
    echo 'APS RESPONSE ERROR: ' . $e->getMessage();
    include '_footer.php';
}
```

Figure 28 Payment method error case

10. Webhook

To handle the request received from the payment gateway you have a method called “getWebhookData” which you can find in “WebhookAdapter” class. This method validates the payment data response and returns it.

```
try {
    $webhookParameters = WebhookAdapter::getWebhookData();
    // your code here
} catch (APSEException $e) {
    Logger::getInstance()->info(
        'Webhook parameters failed to validate! (' . $e->getMessage() . ')'
    );
}
```

Figure 29 Webhook usage

11. Error codes

Error code	Description
1001	APS server to server call failed
1002	APS server to server call response signature failed
1003	APS parameter missing
1004	APS payment adapter missing
1005	APS template file missing
1006	APS callback missing
1007	APS token name missing
1008	APS response signature missing
1009	APS payment method not available
1010	APS invalid type
1011	APS invalid parameter
2001	Apple Pay url missing
2002	Apple Pay url invalid
2003	Apple Pay validation callback url missing
2004	Apple Pay command callback url missing
3001	Response no signature found
4001	Merchant config missing
4002	Merchant config merchant id missing
4003	Merchant config access code missing
4004	Merchant config sha request phrase missing
4005	Merchant config sha response phrase missing
4006	Merchant config sha type missing
5001	Payment data config missing
5002	Payment data merchant reference missing
5003	Payment data amount missing

5004	Payment data currency code missing
5005	Payment data language missing
5006	Payment data customer email missing
5007	Payment data country code missing
5008	Payment data subtotal missing
5009	Payment data shipping missing
5010	Payment data discount missing
5011	Payment data tax missing
6001	Merchant config apple merchant id missing
6002	Merchant config apple access code missing
6003	Merchant config apple supported networks missing
6004	Merchant config apple supported countries missing
6005	Merchant config apple sha request phrase missing
6006	Merchant config apple sha response phrase missing
6007	Merchant config apple sha type missing
6008	Merchant config apple display name missing
6009	Merchant config apple domain name missing
6010	Merchant config apple certificate path missing
6011	Merchant config apple certificate key path missing
6012	Merchant config apple certificate key pass missing
6013	Merchant config sandbox not specified
7001	Webhook parameters empty
7002	Webhook json invalid
7003	Webhook signature invalid