Merchant Integration Guide

PHP API

v. 1.1.0

Moneris SOLUTIONS

Table of Contents

**** PLEASE READ CAREFULLY****

You have a responsibility to protect cardholder and merchant related confidential account information. Under no circumstances should ANY confidential information be sent via email while attempting to diagnose integration or production issues. When sending sample files or code for analysis by Moneris staff, all references to valid card numbers, merchant accounts and transaction tokens should be removed and or obscured. Under no circumstances should live cardholder accounts be used in the test environment.

1. About this Documentation

This document describes the basic information for using the PHP API for sending credit card transactions. In particular, it describes the format for sending transactions and the corresponding responses you will receive. If you are interested in also being able to accept INTERAC payments via your online application, please refer to the PHP API with INTERAC Online Payment document found at:
https://www3.moneris.com/connect/en/documents/index.html

2. System and Skill Requirements

In order to use the PHP API your system will need to have the following:
1. PHP 4 or later
2. Port 443 open
3. OpenSSL
4. cURL - PHP interface - this can be downloaded from http://curl.haxx.se/download.html

As well, you will need to have the following knowledge and/or skill set:
1. PHP programming language

3. What is the Process I will need to follow?

You will need to follow these steps.

1. Do the required development as outlined in this document
2. Test your solution in the test environment
3. Activate your store
4. Make the necessary changes to move your solution from the test environment into production as outlined in this document

4. Transaction Types and Transaction Flow

eSELECTplus supports a wide variety of transactions through the API. Below is a list of transactions supported by the API, other terms used for the transaction type are indicated in brackets.

Purchase – (sale) The Purchase transaction verifies funds on the customer's card, removes the funds and readies them for deposit into the merchant's account.

PreAuth – (authorisation / preauthorisation) The PreAuth verifies and locks funds on the customer's credit card. The funds are locked for a specified amount of time, based on the card issuer. To retrieve the funds from a PreAuth so that they may be settled in the merchant's account a Capture must be performed.

Capture – (Completion / PreAuth Completion) Once a PreAuth is obtained the funds that are locked need to be retrieved from the customer's credit card. The Capture retrieves the locked funds and readies them for settlement into the merchant's account.

Void – (Correction / Purchase Correction) Purchases and Captures can be voided the same day* that they occur. A Void must be for the full amount of the transaction and will remove any record of it from the cardholder's statement.

Refund – (Credit) A Refund can be performed against a Purchase or a Capture to refund any part, or all of the transaction.

Batch Close – (End of Day / Settlement) When a batch close is performed it takes the monies from all Purchase, Capture and Refund transactions so they will be deposited or debited the following business day. For funds to be deposited the following business day the batch must close before 11pm EST.

* A Void can be performed against a transaction as long as the batch that contains the original transaction remains open. When using the automated closing feature Batch Close occurs daily between 10 – 11 pm EST.

5. Transaction Examples

Included below is the sample code that can be found in the "Examples" folder of the PHP API download.

Purchase (basic)

In the purchase example we require several variables (store_id, api_token, order_id, amount, pan, expdate, and crypt_type). Please refer to Appendix A for variable definitions.

```
<?php
// ------ Requires the actual API file.  This can be placed anywhere as long as you indicate
// ------ the proper path

require "../mpgClasses.php";

// ------ Define all the required variables.  These can be passed by whatever means you wish

$store_id = 'store1';
$customerid = 'student_number';
$api_token = 'yesguy';
$orderid = 'need_unique_orderid';
$pan = '5454545454545454';
$amount = '12.00';
$expirydate = '0612';
$crypttype = '7';

// ------ step 1) create transaction hash
$txnArray=array('type'=>'purchase',
         'order_id'=>$orderid,
         'cust_id'=>$customerid,
          'amount'=>$amount,
         'pan'=>$pan,
         'expdate'=>$expirydate,
         'crypt_type'=>$crypttype
           );
// ------ step 2) create a transaction object passing the hash created in step 1.
$mpgTxn = new mpgTransaction($txnArray);

// ------ step 3) create a mpgRequest object passing the transaction object created in step 2
$mpgRequest = new mpgRequest($mpgTxn);

// ------ step 4) create mpgHttpsPost object which does an https post
$mpgHttpPost  =new mpgHttpsPost($store_id,$api_token,$mpgRequest);

// ------ step 5) get an mpgResponse object
$mpgResponse=$mpgHttpPost->getMpgResponse();

// ------ step 6) retrieve data using get methods.  Using these methods you can retrieve the
// ------ appropriate variables (getResponseCode) to check if the transactions is approved
// ------ (=>0 or <50) or declined (>49) or incomplete (NULL)
print ("\nCardType = " . $mpgResponse->getCardType());
print("\nTransAmount = " . $mpgResponse->getTransAmount());
print("\nTxnNumber = " . $mpgResponse->getTxnNumber());
print("\nReceiptId = " . $mpgResponse->getReceiptId());
print("\nTransType = " . $mpgResponse->getTransType());
print("\nReferenceNum = " . $mpgResponse->getReferenceNum());
print("\nResponseCode = " . $mpgResponse->getResponseCode());
print("\nISO = " . $mpgResponse->getISO());
print("\nMessage = " . $mpgResponse->getMessage());
print("\nAuthCode = " . $mpgResponse->getAuthCode());
print("\nComplete = " . $mpgResponse->getComplete());
print("\nTransDate = " . $mpgResponse->getTransDate());
print("\nTransTime = " . $mpgResponse->getTransTime());
print("\nTicket = " . $mpgResponse->getTicket());
print("\nTimedOut = " . $mpgResponse->getTimedOut());

?>
```

PreAuth (basic)

The PreAuth is virtually identical to the Purchase with the exception of the transaction type. It is 'preauth' instead of 'purchase'. Like the Purchase example, PreAuth's require several variables (store_id, api_token, order_id, amount, pan, expdate, and crypt_type). Please refer to Appendix A for variable definitions. In the example below the test environment store_id (store1) and api_token (yesguy) have been hard coded, and highlighted in the mpgHttpsPost object.

```
<?php
// ------ Requires the actual API file.  This can be placed anywhere as long as you indicate
// ------ the proper path

require "../mpgClasses.php";

// ------ Define all the required variables.  These can be passed by whatever means you wish

$orderid = 'need_unique_orderid';
$pan = '5454545454545454';
$amount = '12.00';
$expirydate = '0612';
$crypttype = '7';
$customerid = 'policy_number';

// ------ step 1) create transaction hash
$txnArray=array('type'=>'preauth',
         'order_id'=>$orderid,
         'cust_id'=>$customerid,
         'amoun't=>$amount,
         'pan'=>$pan,
         'expdate'=>$expirydate,
         'crypt_type'=>$crypttype
           );

// ------ step 2) create a transaction  object passing the hash created in step 1.
$mpgTxn = new mpgTransaction($txnArray);

// ------ step 3) create a mpgRequest object passing the transaction object created in step 2
$mpgRequest = new mpgRequest($mpgTxn);

// ------ step 4) create mpgHttpsPost object which does an https post
$mpgHttpPost = new mpgHttpsPost('store1','yesguy',$mpgRequest);

// ------ step 5) get an mpgResponse object
$mpgResponse=$mpgHttpPost->getMpgResponse();

// ------ step 6) retrieve data using get methods.  Using these methods you can retrieve the
// ------ appropriate variables (getResponseCode) to check if the transactions is approved
// ------ (=>0 or <50) or declined (>49) or incomplete (NULL)

print ("\nCardType = " . $mpgResponse->getCardType());
print("\nTransAmount = " . $mpgResponse->getTransAmount());
print("\nTxnNumber = " . $mpgResponse->getTxnNumber());
print("\nReceiptId = " . $mpgResponse->getReceiptId());
print("\nTransType = " . $mpgResponse->getTransType());
print("\nReferenceNum = " . $mpgResponse->getReferenceNum());
print("\nResponseCode = " . $mpgResponse->getResponseCode());
print("\nISO = " . $mpgResponse->getISO());
print("\nMessage = " . $mpgResponse->getMessage());
print("\nAuthCode = " . $mpgResponse->getAuthCode());
print("\nComplete = " . $mpgResponse->getComplete());
print("\nTransDate = " . $mpgResponse->getTransDate());
print("\nTransTime = " . $mpgResponse->getTransTime());
print("\nTicket = " . $mpgResponse->getTicket());
print("\nTimedOut = " . $mpgResponse->getTimedOut());


?>
```

Capture

The Capture (completion) transaction is used to secure the funds locked by a PreAuth transaction. When sending a 'completion' request you will need two pieces of information from the original PreAuth – the order_id and the txn_number from the returned response.

```
<?php

require "../mpgClasses.php";

$store_id='store1';
$api_token='yesguy';
$orderid='need_order_id_from_original_preauth';  //must be from original preauth
$txnnumber='123456-98-0'; //must be from original preauth
$amount='1.00';  //the amount to be captured - cannot be more than 115% of original preauth

// ------ step 1) create transaction array ###
$txnArray=array('type'=>'completion',
         'txn_number'=>$txnnumber,
         'order_id'=>$orderid,
         'comp_amount'=>$amount,
         'crypt_type'=>'7'
           );


// ------ step 2) create a transaction  object passing the hash created in step 1.
$mpgTxn = new mpgTransaction($txnArray);

// ------ step 3) create a mpgRequest object passing the transaction object created in step 2
$mpgRequest = new mpgRequest($mpgTxn);

// ------ step 4) create mpgHttpsPost object which does an https post
$mpgHttpPost  =new mpgHttpsPost($store_id,$api_token,$mpgRequest);

// ------ step 5) get an mpgResponse object
$mpgResponse=$mpgHttpPost->getMpgResponse();

// ------ step 6) retrieve data using get methods.  Using these methods you can retrieve the
// ------ appropriate variables (getResponseCode) to check if the transactions is approved
// ------ (=>0 or <50) or declined (>49) or incomplete (NULL)

print ("\nCardType = " . $mpgResponse->getCardType());
print("\nTransAmount = " . $mpgResponse->getTransAmount());
print("\nTxnNumber = " . $mpgResponse->getTxnNumber());
print("\nReceiptId = " . $mpgResponse->getReceiptId());
print("\nTransType = " . $mpgResponse->getTransType());
print("\nReferenceNum = " . $mpgResponse->getReferenceNum());
print("\nResponseCode = " . $mpgResponse->getResponseCode());
print("\nISO = " . $mpgResponse->getISO());
print("\nMessage = " . $mpgResponse->getMessage());
print("\nAuthCode = " . $mpgResponse->getAuthCode());
print("\nComplete = " . $mpgResponse->getComplete());
print("\nTransDate = " . $mpgResponse->getTransDate());
print("\nTransTime = " . $mpgResponse->getTransTime());
print("\nTicket = " . $mpgResponse->getTicket());
print("\nTimedOut = " . $mpgResponse->getTimedOut());


?>
```

Void

The Void (purchasecorrection) transaction is used to cancel a transaction that was performed in the current batch. No amount is required because a Void is always for 100% of the original transaction. The only transactions that can be Voided are Captures and Purchases. To send a 'purchasecorrection' the order_id and txn_number from the Capture or Purchase are required.

```
<?
require "../mpgClasses.php";

$store_id= 'store1';
$api_token= 'yesguy';
$orderid= 'need_original_order_id';
$txnnumber= '987654-01-1'; //need TxnNumber from capture or purchase transaction that is to be voided

## step 1) create transaction hash ###
$txnArray=array('type'=>'purchasecorrection',
         'txn_number'=>$txnnumber,
         'order_id'=>$orderid,
         'crypt_type'=>'7'
        );

// ------ step 2) create a transaction  object passing the array created in step 1.
$mpgTxn = new mpgTransaction($txnArray);

// ------ step 3) create a mpgRequest object passing the transaction object created in step 2
$mpgRequest = new mpgRequest($mpgTxn);

// ------ step 4) create mpgHttpsPost object which does an https post ##
$mpgHttpPost  =new mpgHttpsPost($store_id,$api_token,$mpgRequest);

// ------ step 5) get an mpgResponse object ##
$mpgResponse=$mpgHttpPost->getMpgResponse();

// ------ step 6) retrieve data using get methods.  Using these methods you can retrieve the
// ------ appropriate variables (getResponseCode) to check if the transactions is approved
// ------ (=>0 or <50) or declined (>49) or incomplete (NULL)

print ("\nCardType = " . $mpgResponse->getCardType());
print("\nTransAmount = " . $mpgResponse->getTransAmount());
print("\nTxnNumber = " . $mpgResponse->getTxnNumber());
print("\nReceiptId = " . $mpgResponse->getReceiptId());
print("\nTransType = " . $mpgResponse->getTransType());
print("\nReferenceNum = " . $mpgResponse->getReferenceNum());
print("\nResponseCode = " . $mpgResponse->getResponseCode());
print("\nISO = " . $mpgResponse->getISO());
print("\nMessage = " . $mpgResponse->getMessage());
print("\nAuthCode = " . $mpgResponse->getAuthCode());
print("\nComplete = " . $mpgResponse->getComplete());
print("\nTransDate = " . $mpgResponse->getTransDate());
print("\nTransTime = " . $mpgResponse->getTransTime());
print("\nTicket = " . $mpgResponse->getTicket());
print("\nTimedOut = " . $mpgResponse->getTimedOut());

?>
```

Refund

The Refund will credit a specified amount to the cardholder's credit card. A Refund can be sent up to the full value of the original Capture or Purchase. To send a 'refund' you will require the order_id and txn_number from the original Capture or Purchase.

```
<?php

require "../mpgClasses.php";

$store_id= 'store1';
$api_token= 'yesguy';
$orderid= 'need_original_order_id';
$txnnumber= '09876-12-0';
$amount = '3.00';

// ------ step 1) create transaction array ###
$txnArray=array('type'=>'refund',
         'txn_number'=>$txnnumber,
         'order_id'=>$orderid,
         'amount'=>$amount,
         'crypt_type'=>'7'
           );

// ------ step 2) create a transaction  object passing the array created in step 1.
$mpgTxn = new mpgTransaction($txnArray);

// ------ step 3) create a mpgRequest object passing the transaction object created in step 2
$mpgRequest = new mpgRequest($mpgTxn);

// ------ step 4) create mpgHttpsPost object which does an https post
$mpgHttpPost  =new mpgHttpsPost($store_id,$api_token,$mpgRequest);

// ------ step 5) get an mpgResponse object ##
$mpgResponse=$mpgHttpPost->getMpgResponse();

// ------ step 6) retrieve data using get methods.  Using these methods you can retrieve the
// ------ appropriate variables (getResponseCode) to check if the transactions is approved
// ------ (=>0 or <50) or declined (>49) or incomplete (NULL)

print ("\nCardType = " . $mpgResponse->getCardType());
print("\nTransAmount = " . $mpgResponse->getTransAmount());
print("\nTxnNumber = " . $mpgResponse->getTxnNumber());
print("\nReceiptId = " . $mpgResponse->getReceiptId());
print("\nTransType = " . $mpgResponse->getTransType());
print("\nReferenceNum = " . $mpgResponse->getReferenceNum());
print("\nResponseCode = " . $mpgResponse->getResponseCode());
print("\nISO = " . $mpgResponse->getISO());
print("\nMessage = " . $mpgResponse->getMessage());
print("\nAuthCode = " . $mpgResponse->getAuthCode());
print("\nComplete = " . $mpgResponse->getComplete());
print("\nTransDate = " . $mpgResponse->getTransDate());
print("\nTransTime = " . $mpgResponse->getTransTime());
print("\nTicket = " . $mpgResponse->getTicket());
print("\nTimedOut = " . $mpgResponse->getTimedOut());

?>
```

Independent Refund

The Independent Refund (ind_refund) will credit a specified amount to the cardholder's credit card. The Independent Refund does not require an existing order to be logged in the eSELECTplus gateway; however, the credit card number and expiry date will need to be passed. In the example below the values are hard coded (and highlighted) in the appropriate locations. The transaction format is almost identical to a Purchase or a PreAuth.

```
<?

require "../mpgClasses.php";

// ------ step 1) create transaction array ###
$txnArray=array('type'=>'ind_refund',
         'order_id'=> 'need_unique order_id',
          'cust_id' => 'need_customer_id',
         'amount'=>'1.00',
         'pan'=>'4242424242424242',
         'expdate'=>'0909',
         'crypt_type'=>'7'
           );

// ------ step 2) create a transaction object passing the array created in step 1.

$mpgTxn = new mpgTransaction($txnArray);

// ------ step 3) create a mpgRequest object passing the transaction object created in step 2
$mpgRequest = new mpgRequest($mpgTxn);

// ------ ## step 4) create mpgHttpsPost object which does an https post ##
$mpgHttpPost = new mpgHttpsPost('store1','yesguy',$mpgRequest);

// ------ step 5) get an mpgResponse object ##
$mpgResponse=$mpgHttpPost->getMpgResponse();

// ------ step 6) retrieve data using get methods.  Using these methods you can retrieve the
// ------ appropriate variables (getResponseCode) to check if the transactions is approved
// ------ (=>0 or <50) or declined (>49) or incomplete (NULL)

print ("\nCardType = " . $mpgResponse->getCardType());
print("\nTransAmount = " . $mpgResponse->getTransAmount());
print("\nTxnNumber = " . $mpgResponse->getTxnNumber());
print("\nReceiptId = " . $mpgResponse->getReceiptId());
print("\nTransType = " . $mpgResponse->getTransType());
print("\nReferenceNum = " . $mpgResponse->getReferenceNum());
print("\nResponseCode = " . $mpgResponse->getResponseCode());
print("\nISO = " . $mpgResponse->getISO());
print("\nMessage = " . $mpgResponse->getMessage());
print("\nAuthCode = " . $mpgResponse->getAuthCode());
print("\nComplete = " . $mpgResponse->getComplete());
print("\nTransDate = " . $mpgResponse->getTransDate());
print("\nTransTime = " . $mpgResponse->getTransTime());
print("\nTicket = " . $mpgResponse->getTicket());
print("\nTimedOut = " . $mpgResponse->getTimedOut());


?>
```

Batch Close

At the end of every day (11pm EST) the Batch needs to be closed in order to have the funds settled the next business day. *By default eSELECTplus will close your Batch automatically for you daily whenever there are funds in the open Batch.* Some merchants prefer to control Batch Close, and disable the automatic functionality. For these merchants we have provided the ability to close your Batch through the API. When a Batch is closed the response will include the transaction count and amount for each type of transaction for each type of card. To disable automatic close you will need to call the technical support line.

```
<?

require "../mpgClasses.php";

$store_id= ‘store1’;
$api_token= ‘yesguy’;

// ------ step 1) create transaction array
$txnArray=array(‘type’=>'batchcloseall'
                );

$mpgTxn = new mpgTransaction($txnArray);

// ------ step 2) create mpgRequest object
$mpgReq=new mpgRequest($mpgTxn);

// ------ step 3) create mpgHttpsPost object which does an https post
$mpgHttpPost=new mpgHttpsPost($store_id,$api_token,$mpgReq);

// ------ step 4) get an mpgResponse object
$mpgResponse=$mpgHttpPost->getMpgResponse();

// ------ step 5) get arrays of all terminal IDs (ecr's) and  credit cards
$ecrs = $mpgResponse->getTerminalIDs();

// ------ step 6) loop through all ecrs
for($x=0; $x < count($ecrs);$x++)
{
 print "\n\nECR Number = $ecrs[$x]";

 $creditCards = $mpgResponse->getCreditCards($ecrs[$x]);

 // ------ loop through the array of credit cards and get information

  for($i=0; $i < count($creditCards); $i++)
   {
     print "\nCard Type = $creditCards[$i]";

     print "\nPurchase Count = "
           . $mpgResponse->getPurchaseCount($ecrs[$x],$creditCards[$i]);
     print "\nPurchase Amount = "
           . $mpgResponse->getPurchaseAmount($ecrs[$x],$creditCards[$i]);
     print "\nRefund Count = "
           . $mpgResponse->getRefundCount($ecrs[$x],$creditCards[$i]);
     print "\nRefund Amount = "
           . $mpgResponse->getRefundAmount($ecrs[$x],$creditCards[$i]);
     print "\nCorrection Count = "
            . $mpgResponse->getCorrectionCount($ecrs[$x],$creditCards[$i]);
     print "\nCorrection Amount = "
           . $mpgResponse->getCorrectionAmount($ecrs[$x],$creditCards[$i]);

    }

 }//end outer for
```

6. Transaction with Extra features - Examples

In the previous section the instructions were provided for the basic transaction set. eSELECTplus also provides several extra features/functionalities. These features include storing customer and order details, Verified by Visa and sending transactions to the Recurring Billing feature. Verified by Visa and Recurring Billing must be added to your account, please call the Service Centre at 1 866 319 7450 to have your profile updated.

Purchase (with Customer and Order details)

Below is an example of sending a Purchase with the customer and order details. If one piece of information is sent then all fields must be included in the request. Unwanted fields need to be blank. The identical format is used for PreAuth with the exception of transaction type which changes from 'purchase' to 'preauth'. Customer details can only be sent with Purchase and PreAuth. It can be used in conjunction with other extra features such as VBV and Recurring Billing. ***Please note that the mpgCustInfo fields are not used for any type of address verification or fraud check.***

```
<?
require "../mpgClasses.php";

$store_id='store1';
$api_token='yesguy';
$orderid= 'need_unique_order_id';


// ------ step 1) create an mpgCustInfo object
$mpgCustInfo = new mpgCustInfo();


// ------ step 2) call set methods of the mpgCustInfo object with the proper information

$email ='Joe@widgets.com';
$mpgCustInfo->setEmail($email);

$instructions ="Make it fast";
$mpgCustInfo->setInstructions($instructions);

$billing = array( 'first_name' => 'Joe',
                  'last_name' => 'Thompson',
                  'company_name' => 'Widget Company Inc.',
                  'address' => '111 Bolts Ave.',
                  'city' => 'Toronto',
                  'province' => 'Ontario',
                  'postal_code' => 'M8T 1T8',
                  'country' => 'Canada',
                  'phone_number' => '416-555-5555',
                  'fax' => '416-555-5555',
                  'tax1' => '123.45',
                  'tax2' => '12.34',
                  'tax3' => '15.45',
                  'shipping_cost' => '456.23');

$mpgCustInfo->setBilling($billing);

$shipping = array( 'first_name' => 'Joe',
                  'last_name' => 'Thompson',
                  'company_name' => 'Widget Company Inc.',
                  'address' => '111 Bolts Ave.',
                  'city' => 'Toronto',
                  'province' => 'Ontario',
                  'postal_code' => 'M8T 1T8',
                  'country' => 'Canada',
                  'phone_number' => '416-555-5555',
                  'fax' => '416-555-5555',
                  'tax1' => '',  // ------ leave blank - use billing above for these values
                  'tax2' => '',  //------ leave blank - use billing above for these values
                  'tax3' => '',  //------ leave blank - use billing above for these values
```

```
                    'shipping_cost' => ''); //------ leave blank - use billing above for these values

$mpgCustInfo->setShipping($shipping);


// ------ set items purchased
$item1 = array ('name'=>'item 2 name',
                'quantity'=>'53',
                'product_code'=>'item 1 product code',
                'extended_amount'=>'1.00');
$mpgCustInfo->setItems($item1);

$item2 = array('name'=>'item 2 name',
                'quantity'=>'53',
                'product_code'=>'item 2 product code',
                'extended_amount'=>'1.00');
$mpgCustInfo->setItems($item2);

// ------ etc...

// ------ step 3) create transaction array ###
$txnArray=array('type'=>'purchase',
         'order_id'=>$orderid,
         'cust_id'=>'optional_field',
         'amount'=>'1.00',
         'pan'=>'4242424242424242',
         'expdate'=>'0909',
         'crypt_type'=>'7'
           );

// ------ step 4) create a transaction object passing the array created in step 3.
$mpgTxn = new mpgTransaction($txnArray);

// ------ step 5) use the setCustInfo method of mpgTransaction object to
// ------ set the customer info (level 3 data) for this transaction
$mpgTxn->setCustInfo($mpgCustInfo);

// ------ step 6) create a mpgRequest object passing the transaction object created  in step 4
$mpgRequest = new mpgRequest($mpgTxn);

// ------ step 7) create mpgHttpsPost object which does an https post
$mpgHttpPost = new mpgHttpsPost($store_id,$api_token,$mpgRequest);

// ------ step 8) get an mpgResponse object ##
$mpgResponse=$mpgHttpPost->getMpgResponse();

// ------  step 9) data using get methods.  Using these methods you can retrieve the
// ------ appropriate variables (getResponseCode) to check if the transactions is approved
// ------ (=>0 or <50) or declined (>49) or incomplete (NULL)

print ("\nCardType = " . $mpgResponse->getCardType());
print("\nTransAmount = " . $mpgResponse->getTransAmount());
print("\nTxnNumber = " . $mpgResponse->getTxnNumber());
print("\nReceiptId = " . $mpgResponse->getReceiptId());
print("\nTransType = " . $mpgResponse->getTransType());
print("\nReferenceNum = " . $mpgResponse->getReferenceNum());
print("\nResponseCode = " . $mpgResponse->getResponseCode());
print("\nISO = " . $mpgResponse->getISO());
print("\nMessage = " . $mpgResponse->getMessage());
print("\nAuthCode = " . $mpgResponse->getAuthCode());
print("\nComplete = " . $mpgResponse->getComplete());
print("\nTransDate = " . $mpgResponse->getTransDate());
print("\nTransTime = " . $mpgResponse->getTransTime());
print("\nTicket = " . $mpgResponse->getTicket());
print("\nTimedOut = " . $mpgResponse->getTimedOut());

?>
```

Purchase (with Verified by Visa – VBV)

Below is an example of sending a Purchase with the Verified by Visa extra fields. The 'cavv' is obtained by using either the Moneris MPI or a third party MPI. The format outlined below is identical for a PreAuth with the exception of the TransType which changes from 'cavv_purchase' to 'cavv_preauth'. VBV must be added to your account, please call the Service Centre at 1 866 319 7450 to have your profile updated. The optional customer and order details can be included in the transaction using steps 1, 2 and 5 from the method above -*Purchase (with Customer and Order Details).*

```
<?
require("mpgClasses.php");

$orderid = 'need_unique_order_id';

$cavv = $mpiResponse->getMpiCavv();  //------ this is a function provided by the Moneris MPI

// ------ Create a transaction array
$txnArray=array(
      'type'=>'cavv_purchase',
      'order_id'=> $orderid,
      'cust_id' => 'optional_customer_id',
      'amount'=>$amount,
      'pan'=>$pan,
      'expdate'=>$expiry,
      'cavv'=>$cavv,
      );

// ------ step 2) create a transaction object passing the array created in step 1.
$mpgTxn = new mpgTransaction($txnArray);

// ------ step 3) create a mpgRequest object passing the transaction object created in step 2
$mpgRequest = new mpgRequest($mpgTxn);

// ------ step 4) create mpgHttpsPost object which does an https post
$mpgHttpPost  =new mpgHttpsPost($store_id,$api_token,$mpgRequest);

// ------ step 5) get an mpgResponse object ##
$mpgResponse=$mpgHttpPost->getMpgResponse();

// ------ step 6) retrieve data using get methods.  Using these methods you can retrieve the
// ------ appropriate variables (getResponseCode) to check if the transactions is approved
// ------ (=>0 or <50) or declined (>49) or incomplete (NULL)

print ("\nCardType = " . $mpgResponse->getCardType());
print("\nTransAmount = " . $mpgResponse->getTransAmount());
print("\nTxnNumber = " . $mpgResponse->getTxnNumber());
print("\nReceiptId = " . $mpgResponse->getReceiptId());
print("\nTransType = " . $mpgResponse->getTransType());
print("\nReferenceNum = " . $mpgResponse->getReferenceNum());
print("\nResponseCode = " . $mpgResponse->getResponseCode());
print("\nISO = " . $mpgResponse->getISO());
print("\nMessage = " . $mpgResponse->getMessage());
print("\nAuthCode = " . $mpgResponse->getAuthCode());
print("\nComplete = " . $mpgResponse->getComplete());
print("\nTransDate = " . $mpgResponse->getTransDate());
print("\nTransTime = " . $mpgResponse->getTransTime());
print("\nTicket = " . $mpgResponse->getTicket());
print("\nTimedOut = " . $mpgResponse->getTimedOut());

?>
```

Purchase (with Recurring Billing)

Recurring Billing is a feature that allows the transaction information to be sent once and then re-billed on a specified interval for a certain number of times. This is a feature commonly used for memberships, subscriptions, or any other charge that is re-billed on a regular basis. The transaction is split into two parts; the recur information and the transaction information. Please see Appendix C for description of each of the fields. The optional customer and order details can be included in the transaction using steps 1, 2 and 5 from the method above -*Purchase (with Customer and Order Details).* Recurring Billing must be added to your account, please call the Service Centre at 1 866 319 7450 to have your profile updated.

```
<?
require "../mpgClasses.php";

// ------ Define all the required variables.  These can be passed by whatever means you wish

$store_id = 'store1';
$custId = 'student_number';

// ------ Recur Variables
$recurUnit = 'day';
$startDate = '2006/11/30';
$numRecurs = '4';
$recurInterval = '10';
$recurAmount = '31.00';
$startNow = 'true';


// ------ Transaction Variables
$api_token = 'yesguy';
$orderId = 'need_unique_orderid';
$creditCard = '5454545454545454';
$nowAmount = '12.00';
$expiryDate = '0712';
$cryptType = '7';

// ------ Create an array with the recur variables
$recurArray = array('recur_unit'=>$recurUnit,  // (day | week | month)
       'start_date'=>$startDate, //yyyy/mm/dd
       'num_recurs'=>$numRecurs,
       'start_now'=>$startNow,
       'period' => $recurInterval,
       'recur_amount'=> $recurAmount
       );

$mpgRecur = new mpgRecur($recurArray);

// ------ step 3) create transaction array ###
$txnArray=array('type'=>'purchase',
       'order_id'=>$orderId,
       'cust_id'=>$custId,
       'amount'=>$nowAmount,
       'pan'=>$creditCard,
       'expdate'=>$expiryDate,
       'crypt_type'=>$cryptType
       );


// ------ step 2) create a transaction object passing the array created in step 1.
$mpgTxn = new mpgTransaction($txnArray);

// ------ step 3) set the Recur Object to mpgRecur
$mpgTxn->setRecur($mpgRecur);

// ------ step 4) create a mpgRequest object passing the transaction object created in step 2
$mpgRequest = new mpgRequest($mpgTxn);

// ------ step 5) create mpgHttpsPost object which does an https post
$mpgHttpPost = new mpgHttpsPost($store_id,$api_token,$mpgRequest);
```

```
// ------ step 6) get an mpgResponse object ##
$mpgResponse=$mpgHttpPost->getMpgResponse();

// ------ step 7) retrieve data using get methods.  Using these methods you can retrieve the
// ------ appropriate variables (getResponseCode) to check if the transactions is approved
// ------ (=>0 or <50) or declined (>49) or incomplete (NULL)

print ("\nCardType = " . $mpgResponse->getCardType());
print("\nTransAmount = " . $mpgResponse->getTransAmount());
print("\nTxnNumber = " . $mpgResponse->getTxnNumber());
print("\nReceiptId = " . $mpgResponse->getReceiptId());
print("\nTransType = " . $mpgResponse->getTransType());
print("\nReferenceNum = " . $mpgResponse->getReferenceNum());
print("\nResponseCode = " . $mpgResponse->getResponseCode());
print("\nISO = " . $mpgResponse->getISO());
print("\nMessage = " . $mpgResponse->getMessage());
print("\nAuthCode = " . $mpgResponse->getAuthCode());
print("\nComplete = " . $mpgResponse->getComplete());
print("\nTransDate = " . $mpgResponse->getTransDate());
print("\nTransTime = " . $mpgResponse->getTransTime());
print("\nTicket = " . $mpgResponse->getTicket());
print("\nTimedOut = " . $mpgResponse->getTimedOut());
print("\nRecurrSuccess = " . $mpgResponse->getRecurSuccess());

?>
```

As part of the Recurring Billing response there will be an additional method called getRecurSuccess( ). This can return a value of 'true' or 'false' based on whether the recurring transaction was successfully registered in our database.

Purchase with CVD and AVS (eFraud)

Below is an example of a Purchase transaction with CVD and AVS information. These values can be sent in conjunction with other additional variables such as Recurring Billing or customer information. With this feature enabled in your merchant profile, you will be able to pass in these fields for the following transactions 'purchase', 'preauth', 'cavv_purchase', and 'cavv_preauth'. To form mpgCvdInfo please refer to Appendix E, to form mpgAvsInfo please refer to Appendix F.

If you wish to have the eFraud feature added to your profile, once you have completed your testing, please contact the eSELECTplus Integration Support Team at eselectplus@moneris.com to have it enabled. Please make sure to include your store ID or merchant number in any eFraud activation email requests.

When testing eFraud (AVS and CVD) you must only use the Visa test card number 4242424242424242 and the amounts described in the Simulator eFraud Response Codes document found at: https://www3.moneris.com/connect/en/documents/index.html
These tests will only work with "store5" as the store_id.

```
<?php
require "../mpgClasses.php";
$store_id=$argv[1];  ##store5
$api_token=$argv[2]; ##yesguy
$orderid=$argv[3];

## step 1) create mpgAVS and mpgCVD objects
$avsTemplate = array('avs_street_number'=>'123',
                     'avs_street_name' =>'East Street',
                     'avs_zipcode' => 'M1M2M2' );
$cvdTemplate = array('cvd_indicator' => '0',
                     'cvd_value' => '123');

$mpgAvsInfo = new mpgAvsInfo ($avsTemplate);
$mpgCvdInfo = new mpgCvdInfo ($cvdTemplate);

## step 2) create transaction array ###
$txnArray=array('type'=>'purchase',
          'order_id'=>$orderid,
           'cust_id'=>'optional_customer_id',
          'amount'=>'1.00',
          'pan'=>'4242424242424242',
          'expdate'=>'0909',
          'crypt_type'=>'7'
            );
## step 3) create a transaction  object passing the array created in step 2.
$mpgTxn = new mpgTransaction($txnArray);

## step 4) use the setAvsInfo and setCvdInfo method of mpgTransaction object to set the eFraud
$mpgTxn->setAvsInfo($mpgAvsInfo);
$mpgTxn->setCvdInfo($mpgCvdInfo);

## step 5) create a mpgRequest object passing the transaction object created in step 3
$mpgRequest = new mpgRequest($mpgTxn);

## step 6) create mpgHttpsPost object which does an https post
$mpgHttpPost  =new mpgHttpsPost($store_id,$api_token,$mpgRequest);

## step 8) get an mpgResponse object ##
$mpgResponse=$mpgHttpPost->getMpgResponse();

## step 9) retrieve data using get methods
print ("\nCardType = " . $mpgResponse->getCardType());
print("\nTransAmount = " . $mpgResponse->getTransAmount());
print("\nTxnNumber = " . $mpgResponse->getTxnNumber());
print("\nReceiptId = " . $mpgResponse->getReceiptId());
print("\nTransType = " . $mpgResponse->getTransType());
print("\nReferenceNum = " . $mpgResponse->getReferenceNum());
print("\nResponseCode = " . $mpgResponse->getResponseCode());
print("\nISO = " . $mpgResponse->getISO());
print("\nMessage = " . $mpgResponse->getMessage());
print("\nAuthCode = " . $mpgResponse->getAuthCode());
print("\nComplete = " . $mpgResponse->getComplete());
```

```
print("\nTransDate = " . $mpgResponse->getTransDate());
print("\nTransTime = " . $mpgResponse->getTransTime());
print("\nTicket = " . $mpgResponse->getTicket());
print("\nTimedOut = " . $mpgResponse->getTimedOut());
print("\nAVSResponse = " . $mpgResponse->getAvsResultCode());
print("\nCVDResponse = " . $mpgResponse->getCvdResultCode());
?>
```

7. What Information will I get as a Response to My Transaction Request?

For each transaction you will receive a response message. For a full description of each field please refer to Appendix B.

To determine whether a transaction is successful or not the field that must be checked is ResponseCode. See the table below to determine the transaction result.

Response Code	Result
0 – 49 (inclusive)	Approved
50 – 999 (inclusive)	Declined
Null	Incomplete

For a full list of response codes and the associated message please refer to https://www3.moneris.com/connect/en/documents/index.html and download the Response Code document.

8. How Do I Test My Solution?

A testing environment is available for you to connect to while you are integrating your site to our payment gateway. The test environment is generally available 7x24, however since it is a test environment we cannot guarantee 100% availability. Also, please be aware that other merchants are using the test environment so you may see transactions and user IDs that you did not create. As a courtesy to others that are testing we ask that when you are processing Refunds, changing passwords and/or trying other functions that you use only the transactions/users that you created.

When using the APIs in the test environment you will need to use test store_id and api_token. These are different than your production IDs. The IDs that you can use in the test environment are in the table below.

Test IDs			
store_id	**api_token**	**Username**	**Password**
store1	yesguy	DemoUser	password
store2	yesguy	DemoUser	password
store3	yesguy	DemoUser	password
store5 *	yesguy	DemoUser	password
moneris **	hurgle	DemoUser	password

* "store5" is for testing eFraud (AVS & CVD)
** the "moneris" store is for testing VBV

When testing you may use the following test card numbers with any future expiry date.

Test Card Numbers	
Card Plan	**Card Number**
MasterCard	5454545454545454
Visa	4242424242424242
Amex	373599005095005
Diners	36462462742008

To access the Merchant Resource Centre in the test environment go to https://esqa.moneris.com/mpg. And use the logins provided in the previous table.

The test environment has been designed to replicate our production environment as closely as possible. One major difference is that we are unable to send test transactions onto the production authorization network and thus Issuer responses are simulated. Additionally, the requirement to emulate approval, decline and error situations dictates that we use certain transaction variables to initiate various response and error situations.

The test environment will approve and decline transactions based on the penny value of the amount field.
For example, a transaction made for the amount of $399.00 or $1.00 will approve since the .00 penny value is set to approve in the test environment. Transactions in the test environment should not exceed $1000.00. This limit does not exist in the production environment. For a list of all current test environment responses for various penny values, please see the Test Environment Penny Response table as well as the Test Environment eFraud Response table, available at https://www3.moneris.com/connect/en/documents/index.html.

NOTE These responses may change without notice. Moneris Solutions recommends you regularly refer to our website to check for possible changes.

9. What Do I Need to Include in the Receipt?

Visa and MasterCard expect certain variables be returned to the cardholder and presented as a receipt when a transaction is approved. These 12 fields are listed below. A sample receipt is provided in Appendix H.

1. Amount
2. Transaction Type
3. Date and Time
4. Auth Code
5. Response Code
6. ISO Code
7. Response Message
8. Reference Number
9. Goods and Services Order
10. Merchant Name
11. Merchant URL
12. Cardholder Name
13. Return Policy (only a requirement for e-commerce transactions)

10. How Do I Activate My Store?

Once you have received your activation letter/fax go to https://www3.moneris.com/connect/en/activate/index.php as instructed in the letter/fax. You will need to input your store ID and merchant ID then click on 'Activate'. In this process you will need to create an administrator account that you will use to log into the Merchant Resource Centre to access and administer your eSELECTplus store. You will need to use the Store ID and API Token to send transactions through the API.

Once you have created your first Merchant Resource Centre user, please log on to the Interface by clicking the "eSELECTplus" button. Once you have logged in please proceed to ADMIN and then STORE SETTINGS. At the bottom please place a check beside the APIs that you are using. This will allow us to keep you up to date regarding any changes to the APIs that may affect your store. Also, this is where you may locate your production API Token.

11. How Do I Configure My Store For Production?

Once you have completed you testing you are ready to point your store to the production host. You will need to edit the mpgClasses.php file and change the $Globals array as highlighted below in red. You will also need to change the store_id to reflect your production store ID as well the api_token must be changed to your production token to reflect the token that you received during activation.

PRODUCTION	DEVELOPMENT
var $Globals=array('MONERIS_PROTOCOL' => 'https', 'MONERIS_HOST' => **'www3.moneris.com'**, 'MONERIS_PORT' =>'443', 'MONERIS_FILE' => '/gateway2/servlet/MpgRequest', 'API_VERSION' =>'MPG Version 2.01', 'CLIENT_TIMEOUT' => '60');	var $Globals=array('MONERIS_PROTOCOL' => 'https', 'MONERIS_HOST' => **'esqa.moneris.com'**, 'MONERIS_PORT' =>'443', 'MONERIS_FILE' => '/gateway2/servlet/MpgRequest', 'API_VERSION' =>'MPG Version 2.01', 'CLIENT_TIMEOUT' => '60');

Once you are in production you will access the Merchant Resource Centre at https://www3.moneris.com/mpg. You can use the store administrator ID you created during the activation process and then create additional users as needed.

For further information on how to use the Merchant Resource Centre please see the eSELECTplus Merchant Resource Centre User's Guide which is available at https://www3.moneris.com/connect/en/documents/index.html.

12. How Do I Get Help?

If you require technical assistance while integrating your store, please contact the eSELECTplus Support Team:

For technical support:
Phone: 1-866-319-7450 (Technical Difficulties)

For integration support:
Phone: 1-866-562-4354
Email: eselectplus@moneris.com

When sending an email support request please be sure to include your name and phone number, a clear description of the problem as well as the type of API that you are using. **For security reasons, please do not send us your API Token combined with your store ID, or your merchant number and device number in the same email.**

13. Appendix A. Definition of Request Fields

Request Fields		
Variable Name	**Size/Type**	**Description**
order_id	50 / an	Merchant defined unique transaction identifier - must be unique for every Purchase, PreAuth and Independent Refund attempt. For Refunds, Completions and Voids the order_id must reference the original transaction.
pan	20 / variable	Credit Card Number - no spaces or dashes. Most credit card numbers today are 16 digits in length but some 13 digits are still accepted by some issuers. This field has been intentionally expanded to 20 digits in consideration for future expansion and/or potential support of private label card ranges.
expdate	4 / num	Expiry Date - format **YYMM** no spaces or slashes. PLEASE NOTE THAT THIS IS REVERSED FROM THE DATE DISPLAYED ON THE PHYSICAL CARD WHICH IS MMYY
amount	9 / decimal	Amount of the transaction. This must contain 3 digits with two penny values. The minimum value passed can be 0.01 and the maximum 9999999.99
crypt_type	1 / an	E-Commerce Indicator: 1 - Mail Order / Telephone Order - Single 2 - Mail Order / Telephone Order - Recurring 3 - Mail Order / Telephone Order - Instalment 4 - Mail Order / Telephone Order - Unknown Classification 5 - Authenticated E-commerce Transaction (VBV) 6 – Non Authenticated E-commerce Transaction (VBV) 7 - SSL enabled merchant 8 - Non Secure Transaction (Web or Email Based) 9 - SET non - Authenticated transaction
txn_number	255 / varchar	Used when performing follow on transactions - this must be filled with the value that was returned as the Txn_number in the response of the original transaction. When performing a Capture this must reference the PreAuth. When performing a Refund or a Void this must reference the Capture or the Purchase.
cust_id	50/an	This is an optional field that can be sent as part of a Purchase or PreAuth request. It is searchable from the Moneris Merchant Resource Centre. It is commonly used for policy number, membership number, student ID or invoice number.
cavv		This is a value that is provided by the Moneris MPI or by a third party MPI. It is part of a VBV transaction.
avs_street_number avs_street_name	19 / an	Street Number & Street Name (max – 19 digit limit for street number and street name combined). This must match the address that the issuing bank has on file.
avs_zipcode	10 / an	Zip or Postal Code – This must match what the issuing banks has on file.
cvd_value	4 / num	Credit Card CVD value – this number accommodates either 3 or 4 digit CVD values.
cvd_indicator	1 / num	CVD presence indicator (1 digit – refer to Appendix E for values)

14. Appendix B. Definitions of Response Fields

Response Fields		
Variable Name	**Size/Type**	**Description**
order_id	50 / an	order_id specified in request
ReferenceNum	18 / num	The reference number is an 18 character string that references the terminal used to process the transaction as well as the shift, batch and sequence number, This data is typically used to reference transactions on the host systems and must be displayed on any receipt presented to the customer. This information should be stored by the merchant. The following illustrates the breakdown of this field where "660123450010690030" is the reference number returned in the message, "66012345" is the terminal id, "001" is the shift number, "069" is the batch number and "003" is the transaction number within the batch. Moneris Host Transaction identifier.
ReponseCode	3 / num	Transaction Response Code < 50: Transaction approved >= 50: Transaction declined NULL: Transaction was not sent for authorization * If you would like further details on the response codes that are returned please see the Response Codes document available at https://www3.moneris.com/connect/en/documents/index.html
ISO	2 / num	ISO response code
AuthCode	8 / an	Authorization code returned from the issuing institution
TransTime	##:##:##	Processing host time stamp
TransDate	yyyy-mm-dd	Processing host date stamp
TransType	an	Type of transaction that was performed
Complete	True/False	Transaction was sent to authorization host and a response was received
Message	100 / an	Response description returned from issuing institution.
TransAmount		
CardType	2 / alpha	Credit Card Type
Txn_number	20 / an	Gateway Transaction identifier
TimedOut	True/False	Transaction failed due to a process timing out
Ticket	n/a	reserved
RecurSucess	True/false	Indicates whether the transaction successfully registered.
AvsResultCode	1/alpha	Indicates the address verification result. Refer to Appendix F.
CvdResultCode	2/an	Indicates the CVD validation result. Refer to Appendix E.

15. Appendix C. Recur Fields

Recur Request Fields		
Variable Name	**Size/Type**	**Description**
recur_unit	day, week, month	The unit that you wish to use as a basis for the Interval. This can be set as day, week or month. Then using the "period" field you can configure how many days, weeks, months between billing cycles.
period	0 – 999 / num	This is the number of recur_units you wish to pass between billing cycles. Example : period = 3, recur_unit=month -> Card will be billed every 3 months. period = 4, recur_unit=weeks -> Card will be billed every 4 weeks. period = 45, recur_unit=day -> Card will be billed every 45 days. Please note that the total duration of the recurring billing transaction should not exceed 5-10 years in the future.
start_date	YYYY/MM/DD	This is the date on which the first charge will be billed. The value must be in the future. It cannot be the day on which the transaction is being sent. If the transaction is to be billed immediately the start_now feature must be set to true and the start_date should be set at the desired interval after today.
start_now	true / false	When a charge is to be made against the card immediately start_now should be set to 'true'. If the billing is to start in the future then this value is to be set to 'false'. When start_now is set to 'true' the amount to be billed immediately may differ from the recur amount billed on a regular basis thereafter.
recur_amount	9 / decimal	Amount of the recurring transaction. This must contain 3 digits with two penny values. The minimum value passed can be 0.01 and the maximum 9999999.99. This is the amount that will be billed on the start_date and every interval thereafter.
num_recurs	1 – 99 / num	The number of times to recur the transaction.
amount	9 / decimal	When start_now is set to 'true' the amount field in the transaction array becomes the amount to be billed immediately. When start_now is set to 'false' the amount field in the transaction array should be the same as the recur_amount field.

Recur Request Examples	
Recur Request Exampl	**Description**

`$recurArray = array('recur_unit'=>'month',`
`'start_date'=>'2007/01/02',`
`'num_recurs'=>'12',`
`'start_now'=>'false',`
`'period' => '2',`
`'recur_amount'=> '30.00'`
`);`
 	
`$mpgRecur = new mpgRecur($recurArray);`
 	
`// ------ step 3) create transaction array ###`
`$txnArray=array('type'=>'purchase',`
`'order_id'=>'monthly_bill',`
`'cust_id'=>'mem-1234-01',`
`'amount'=>'30.00',`
`'pan'=>'5454545454545454',`
`'expdate'=>'0712',`
`'crypt_type'=>'2'`
`);`
	In the example to the left the first transaction will occur in the future on Jan 2nd 2007. It will be billed $30.00 every 2 months on the 2nd of each month. The card will be billed a total of 12 times.

```
$recurArray = array('recur_unit'=>'week',
        'start_date'=>'2007/01/02',
        'num_recurs'=>'26',
        'start_now'=>'true',
        'period' => '2',
        'recur_amoun't=> '30.00'
        );

$mpgRecur = new mpgRecur($recurArray);

// ------ step 3) create transaction array ###
$txnArray=array('type'=>'purchase',
        'order_id'=>'biweekly_bill',
        'cust_id'=>'mem-1234-02',
        'amount'=>'15.00',
        'pan'=>'5454545454545454',
        'expdate'=>'0712',
        'crypt_type'=>'2'
        );
```

In the example on the left the first charge will be billed immediately. The initial charge will be for $15.00. Then starting on Jan 2nd 2007 the credit card will be billed $30.00 every 2 weeks for 26 recurring charges. The card will be billed a total of 27 times. (1 x $15.00 (immediate) and 26 x $30.00 (recurring))

NOTE

When completing the recurring billing portion please keep in mind that to prevent the shifting of recur bill dates, avoid setting the start_date for anything past the 28th of any given month. For example, all billing dates set for the 31st of May will shift and bill on the 30th in June and will then bill the cardholder on the 30th for every subsequent month.

16. Appendix D. Error Messages

Global Error Receipt – You are not connecting to our servers. This can be caused by a firewall or your internet connection.

Response Code = NULL – The response code can be returned as null for a variety of reasons. A majority of the time the explanation is contained within the Message field. When a 'NULL' response is returned it can indicate that the Issuer, the credit card host, or the gateway is unavailable, either because they are offline or you are unable to connect to the internet. A 'NULL' can also be returned when a transaction message is improperly formatted.

Below are error messages that are returned in the Message field of the response.

Message: XML Parse Error in Request: <System specific detail>
Cause: For some reason an improper XML document was sent from the API to the servlet

Message: XML Parse Error in Response: <System specific detail>
Cause: For some reason an improper XML document was sent back from the servlet

Message: Transaction Not Completed Timed Out
Cause: Transaction times out before the host responds to the gateway

Message: Request was not allowed at this time
Cause: The host is disconnected

Message: Could not establish connection with the gateway:
<System specific detail>
Cause: Gateway is not accepting transactions or server does not have proper access to internet

Message: Input/Output Error: <System specific detail>
Cause: Servlet is not running

Message: The transaction was not sent to the host because of a duplicate order id
Cause: Tried to use an order id which was already in use

Message: The transaction was not sent to the host because of a duplicate order id
Cause: Expiry Date was sent in the wrong format

17. Appendix E. Card Validation Digits (CVD)

The Card Validation Digits (CVD) value refers to the numbers appearing on the back of the credit card which are not imprinted on the front. The exception to this is with American Express cards where this value is indeed printed on the front. The mpgCvdInfo parameter is broken down into two elements. The first element is the CVD Value itself.

The second element is the CVD Indicator. This value indicates the possible scenarios when collecting CVD information. This is a 1 digit value which can have any of the following values:

CVD Indicator:
0 = CVD value is deliberately bypassed or is not provided by the merchant.
1 = CVD value is present.
2 = CVD value is on the card, but is illegible.
9 = Cardholder states that the card has no CVD imprint.

CVD Response codes:

The CVD response is an alphanumeric 2 byte variable. The first byte is the numeric CVD indicator sent in the request; the second byte would be the response code. The following is a list of all possible responses once a CVD value has been passed in.

CVD Response Code:
M = Match
N = No Match
P = Not Processed
S = CVD should be on the card, but Merchant has indicated that CVD is not present
U = Issuer is not a CVD participant


NOTE

For American Express, a CVD Response code will not be returned; the response will be either Approved or Declined.

To have CVD for American Express added to your profile, please contact AmEx directly.

***For additional information on how to handle these responses, please refer to Appendix G.**

18. Appendix F. Address Verification Service (AVS)

The Address Verification Service (AVS) value refers to the cardholder's street number, street name and zip/postal code as it would appear on their statement. mpgAvsInfo is broken down into three elements:

Element	Type	Length
Street Number	Numeric	19 characters combined.
Street Name	Alphanumeric	
Zip/Postal Code	Alphanumeric	9 characters

The following table outlines the possible responses when passing in AVS information.

Table of possible Visa and MC AVS result codes			
VALUE	**DESCRIPTION**	**Domestic**	**International**
A	Street addresses match. The street addresses match but the postal/ZIP codes do not, or the request does not include the postal/ZIP code.	√	√
B	Street addresses match. Postal code not verified due to incompatible formats. (Acquirer sent both street address and postal code.)	√	√
C	Street address and postal code not verified due to incompatible formats. (Acquirer sent both street address and postal code.)	√	√
D	Street addresses and postal codes match.		√
G	Address information not verified for international transaction.		√
I	Address information not verified.		√
M	Street address and postal code match.		√
N	No match. Acquirer sent postal/ZIP code only, or street address only, or both postal code and street address.	√	√
P	Postal code match. Acquirer sent both postal code and street address, but street address not verified due to incompatible formats.	√	√
R	Retry: System unavailable or timed out. Issuer ordinarily performs its own AVS but was unavailable. Available for U.S. issuers only.	√	
S	Not applicable. If present, replaced with G (for international) or U (for domestic) by V.I.P. Available for U.S. Issuers only.	√	
U	Address not verified for domestic transaction. Visa tried to perform check on issuer's behalf but no AVS information was available on record, issuer is not an AVS participant, or AVS data was present in the request but issuer did not return an AVS result.	√	
W	Not applicable. If present, replaced with "Z" by V.I.P. Available for U.S. issuers only.	√	
X	Not applicable. If present, replaced with "Y" by V.I.P. Available for U.S. issuers only.	√	
Y	Street address and postal code match.	√	
Z	Postal/ZIP matches; street address does not match or street address not included in request.	√	√

Table of possible AVS response codes from Discover	
VALUE	**DESCRIPTION**
X	All digits match, nine-digit zip code.
A	All digits match, five-digit zip code.
Y	Address matches, zip code does not match.
T	Nine-digit zip code matches, address does not match.
Z	Five-digit zip codes matches, address does not match.
N	Nothing matches.
W	No data from issuer/authorization system.
U	Retry, system unable to process.
S	AVS not supported at this time.

19. Appendix G. Additional Information for CVD and AVS

The responses that are received from CVD and AVS verifications are intended to provide added security and fraud prevention, but the response itself will not affect the completion of a transaction. Upon receiving a response, the choice to proceed with a transaction is left entirely to the merchant.

Please note that all responses coming back from these verification methods are not direct indicators of whether a merchant should complete any particular transaction. The responses should not be used as a strict guideline of which transaction will approve or decline.


NOTE

Please note that CVD verification is only applicable towards Visa, MasterCard and AmEx transactions.

Also, please note that AVS verification is only applicable towards Visa, MasterCard, and Discover transactions. This verification method is not applicable towards AmEx or any other card type.

20. Appendix H. Sample Receipt

Your order has been Approved
Print this receipt for your records

QA Merchant #1
3250 Bloor St West
Toronto Ontario
M8X2X9

1 800 987 1234
www.moneris.com

Transaction Type: Purchase

Order ID:	mhp3495435587		
Date/Time:	2002-10-18 11:27:48	Approval Code:	030012
Sequence Number:	660021630012090020	Response / ISO Code:	028/04
Amount:	12.00	APPROVED * =	

Item	Description	Qty	Amount	Subtotal
cir-001	Med Circle	1	2.00	2.00
tri-002	Big triangle	1	1.00	1.00
squ-003	small square	2	1.00	3.00
			Shipping:	4.00
			GST :	1.00
			PST :	1.00
			Total:	12.00 CAD

Bill To:

Test Customer

123 Main St
Springfield
ON
Canada
M1M 1M1
tel: 416 555 1111
fax: 416 555 1111

Ship To:

Test

1 King St
Bakersville
ON
Canda
M1M 1M1
tel: 416 555 2222
fax: 416 555 2222

Special Instructions

Knock on Back door when delivering
E-Mail Address:eselectsupport@moneris.com

Refund Policy

30 Days - Must be unopened, 10% restocking charge.