API – Server Post

Public - Security Level 0

March 2022

REVISION HISTORY

Date	Version	Author(s)	Comments
5/6/16	2.0	C. Meaney	Reformatting
10/25/2021	2.1	M. Billips	Bring current, slight modification to code samples

CONFIDENTIALITY STATEMENT

Contents

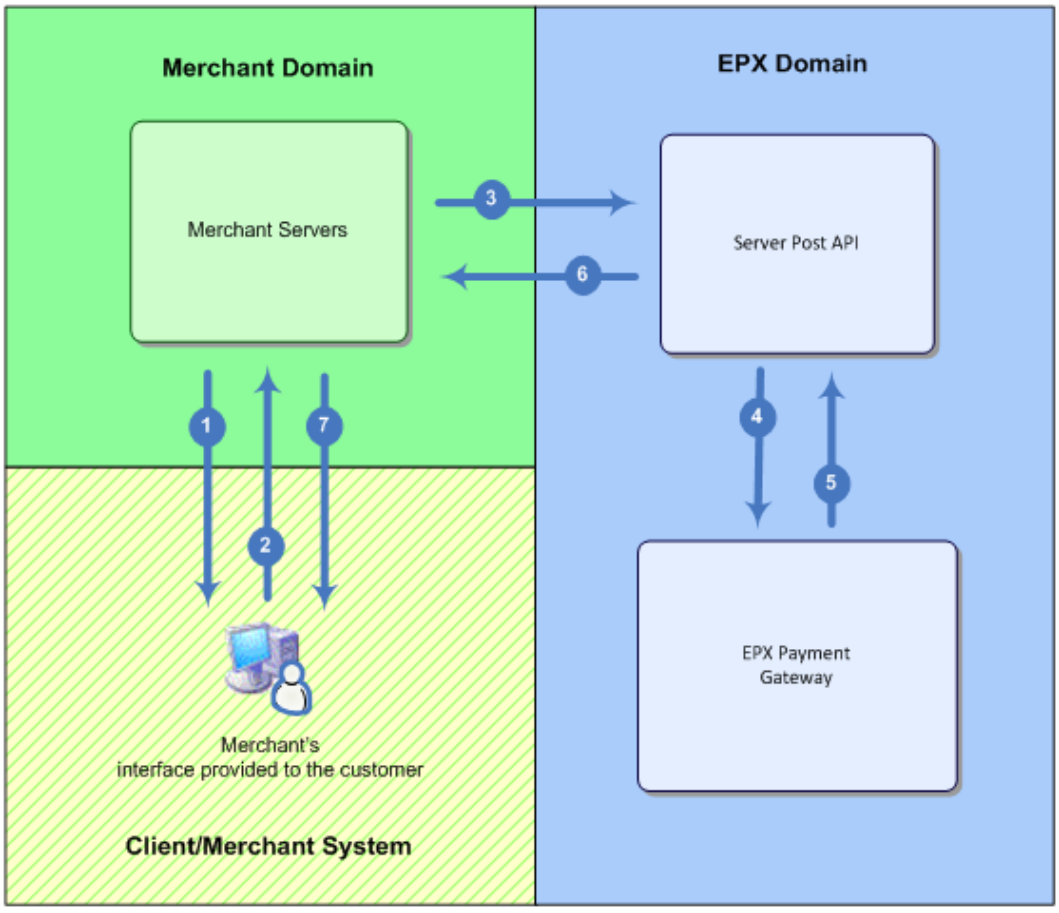
Server Post introduction

Overview

The EPX Server Post API is a standard HTTP service designed to allow the secure integration of financial transactions from a merchant's server directly to the EPX Payment Gateway. When implemented correctly, a merchant can send any transaction type from an agent- or customer-driven interface, through their servers, directly to EPX for real-time financial processing. This most basic method of connecting to EPX is used to process transactions containing full customer account information, as well as using the EPX token (known as a BuyerWall recognized identification code or "BRIC"). The implementation of the Server Post is commonly used for merchant-driven recurring BRIC transactions after a BRIC has been established for a customer account through the EPX PayPage or Browser Post API. This is just one of the common uses for the Server Post API that can assist in allowing the merchant to meet all of their processing needs with the EPX token technology.

Process flow

The figure that follows shows the EPX Server Post processing flow.



The following sequence describes the flow shown in the figure:

1. The merchant provides an interface to take the card information from the agent or customer.
2. The card and customer information are taken from the interface and delivered back to the merchant's servers.
3. The merchant either sends the HTTPS Post transaction request, or creates the XML request string and opens a secure channel via a specified port to send the request data to the Server Post API.
4. The Server Post API formats the financial transaction and submits it to the EPX Payment Gateway.
5. The EPX Payment Gateway responds back to the Server Post API with the network response information.
6. The transaction response information is formatted in XML and sent on the same connection back to the merchant's server typically within a second of the request being received.
7. The merchant's server can then parse the XML results to determine if the response is an approval or decline, and then provide this information back to the originating interface.

Server Post request

Overview

Two options are available for posting a request to the Server Post API:

- Using an HTTPS POST method that follows the standard key-value pair format.
- Using an XML formatted request that is sent over an alternate secure port.

Both methods are discussed in detail in this section. The *EPX Data Dictionary* document defines each of the possible transaction fields that can be sent to EPX during a transaction, and the *EPX Transaction Specifications* document provide examples of the most common transaction types and the fields that are required for each.

NOTE: Authentic Sites use SSL Web Server Certificates to offer secure communications by encrypting all data going to and from the site. The Certificate Authority (CA) has checked and verified the company registration documents and the site's registered domain name. This information is included in the SSL certificate issued. This enables you to check the site's validity yourself. Always check a site's certificate before entering any sensitive information.

If any SSL certificate errors are encountered when connecting to EPX, the transaction should not be sent.

HTTPS POST request method

When using this method the data is formatted utilizing the key-value pair format and sent via a POST request method to the Server Post API over the standard SSL port, 443. The data stream should look similar to the following sample data:

```
CUST_NBR=1234&MERCH_NBR=123456&DBA_NBR=1&TERMINAL_NBR…..
```

The key portion of each pair represents the field name from the EPX Data Dictionary, while the value portion represents the merchant's data that will be the value for that field. In the sample provided, the "CUST_NBR" is a key or EPX field, and "1234" is the value or merchant's data.

Each key-value pair is separated by an ampersand (&), and an End of Request marker is not required because the last field is simply the end of the request. When submitting a request in this manner, the fields do not need to be in any particular order.

When using an HTTPS POST method, an individual connection to the Server Post API is established for each transaction request and the connection is closed after the response is received.

Please refer to the *EPX Transaction Specifications* for example HTTPS POST requests using the most common transaction types.

XML secure socket method

When using this method the data is formatted utilizing the EPX proprietary XML format and then sent through an SSL stream that is established over a TCP connection to the alternate secure port, 8086. The merchant submits the XML request and receives a response in XML format, typically within a second. The XML request stream should look similar to the following sample data:

```
<DETAIL CUST_NBR="1234" MERCH_NBR="1234567" DBA_NBR="1" TERMINAL_NBR="1">
<TRAN_TYPE>CCE1</TRAN_TYPE>
<AMOUNT>11.95</AMOUNT>
<ACCOUNT_NBR>4111111111111111</ACCOUNT_NBR>
.
.
.
</DETAIL>
```

The root element of the XML stream is the DETAIL tag, which contains the merchant's four-part processing key. The key is defined as attributes to that node. All other transaction data is formatted as child elements of the root DETAIL node.

All transaction sub-elements are formatted as simple content with no attributes and may occur in the XML stream in no particular order.

Unlike the HTTPS POST method, the merchant is responsible for closing the TCP socket connection after sending and receiving the transaction response. EPX will keep the connection open for a short period, allowing multiple transactions to be sent, but will close the connection after 30 seconds of inactivity. Merchants might find it useful to integrate using this persistent connection method when high transaction volume is expected, or in locations where the Internet connection is not as stable and requires monitoring.

Please refer to the *EPX Transaction Specifications* for example XML requests using the most common transaction types.

Server Post response

Overview

After sending the Request to EPX, you will receive a Response in XML format that contains the fields that indicate the results of the Request. Please refer to the *EPX Data Dictionary* for a list of possible response fields and definitions for each.

NOTE: The XML response should always be parsed as XML, when the merchant is coding to accept the request and determine the outcome of the transaction. If parsed in a different manner, the parsing procedure results might not be as expected if the response fields are in a different order. Though EPX attempts to provide a similar order each time, the order cannot be guaranteed due to the issuer or card network's response potentially changing with each transaction.

Below is an example of a basic Server Post XML Response. The Response format will always be consistent, however different profile settings and the implementation of new features could change which fields are returned. Enhanced response fields can be made available upon request in the test environment; however this response data is canned and is provided only to show basic examples of the additional fields being included in the response.

Response fields can be returned in the response by enabling the Verbose Response field that is defined in the *EPX Data Dictionary* document.

```
<RESPONSE>
<FIELDS>
<FIELD KEY="MSG_VERSION">003</FIELD>
<FIELD KEY="CUST_NBR">1234</FIELD>
<FIELD KEY="MERCH_NBR">123456</FIELD>
<FIELD KEY="DBA_NBR">1</FIELD>
<FIELD KEY="TERMINAL_NBR">1</FIELD>
<FIELD KEY="TRAN_TYPE">CCE1</FIELD>
<FIELD KEY="BATCH_ID">20130708</FIELD>
<FIELD KEY="TRAN_NBR">20</FIELD>
<FIELD KEY="LOCAL_DATE">070813</FIELD>
<FIELD KEY="LOCAL_TIME">140612</FIELD>
<FIELD KEY="AUTH_GUID">09MBFZ3PV6BTRETVQK2</FIELD>
<FIELD KEY="AUTH_RESP">00</FIELD>
<FIELD KEY="AUTH_CODE">001084</FIELD>
<FIELD KEY="AUTH_AVS">Y</FIELD>
<FIELD KEY="AUTH_CVV2">M</FIELD>
<FIELD KEY="AUTH_RESP_TEXT">EXACT MATCH</FIELD>
<FIELD KEY="AUTH_CARD_TYPE">V</FIELD>
<FIELD KEY="AUTH_TRAN_DATE_GMT">07/08/2013 08:06:12 PM</FIELD>
</FIELDS>
```

```
</RESPONSE>
```

Sample source code

Overview

This section contains sample code of different languages that can be used as a reference for building applications that will send transaction requests to the Server Post API.

HTTPS POST code examples

HTML

```
<html>
<head>
</head>
<body>
<h1>Sample HTML Post</h1>
<h2>CC Transaction</h2>
<form action='https://test.test.com' method="POST">
<table>
<tr>
<td valign="top">
CUST_NBR: <input type="hidden"name="CUST_NBR" value="1234"/><br>
MERCH_NBR: <input type="text" name="MERCH_NBR" value="1234567"/><br>
DBA_NBR: <input type="text" name="DBA_NBR" value="1"/><br> TERMINAL_NBR:
<input type="text" name="TERMINAL_NBR" value="1"/><br>
BATCH_ID: <input type="text" name="BATCH_ID" value="5"/><br> TRAN_NBR:
<input type="text" name="TRAN_NBR" value="9"/><br> AMOUNT: <input
type="text" name="AMOUNT" value="11.99"/><br>
ACCOUNT_NBR: <input type="text" name="ACCOUNT_NBR"
value="4111111111111111"/><br>
EXP_DATE: <input type="text" name="EXP_DATE" value="2709"/><br> TRAN_TYPE:
<input type="text" name="TRAN_TYPE" value="CCM1"/><br>
```

PHP

```
<?php

// Build the request to send via the post function buildRequest() {

$request="CUST_NBR=1234&";
$request.="MERCH_NBR=1234567&";
$request.="DBA_NBR=1&";
$request.="TERMINAL_NBR=1&";

// credit card
$request.="TRAN_TYPE=CCM1&";
$request.="ACCOUNT_NBR=4111111111111111&";
$request.="EXP_DATE=2708&";
$request.="CARD_ENT_METH=X&";
$request.="BATCH_ID=1200&";
$request.="TRAN_NBR=4&";
$request.="AMOUNT=15.01";

return $request;
}

// Send the request via curl function sendRequest($request) {
$ch=curl_init();
curl_setopt($ch, CURLOPT_URL,"https://test.test.com"); curl_setopt($ch,
CURLOPT_POST, 1);
curl_setopt($ch, CURLOPT_POSTFIELDS, $request); curl_setopt($ch,
CURLOPT_RETURNTRANSFER, 1);

$response=curl_exec ($ch); if (curl_errno($ch)!=0) echo curl_error($ch);
curl_close($ch);
return $response;
}
// Build, send, and receive a transaction
$request = buildRequest();
$response=sendRequest($request);
// Print out the whole response string for testing echo($response);

?>
<html>
<head>
</head>
<body>
<h2>Sample HTML/PHP Post</h2>
<form action="sample.php" method="POST">
</form>
```

```
</body>
</html>
```

.NET

```
using System;
using System.Collections;
using System.Web;
using System.Net;
using System.IO;
using System.Text;

namespace WebApplication1
{
/// <summary>
/// Summary description for WebForm1.
/// </summary>
public partial class WebForm1 : System.Web.UI.Page
{

protected System.Web.UI.WebControls.TextBox TextBox1;
protected System.Web.UI.WebControls.Label Label1;
protected System.Web.UI.WebControls.Button Button1;

private void Page_Load(object sender, System.EventArgs e)
{
// Put user code to initialize the page here
}

Web Form Designer generated code

private void Button1_Click(object sender, System.EventArgs e)
{
ASCIIEncoding encoding=new ASCIIEncoding(); string postData =
"CUST_NBR=" + HttpUtility.UrlEncode("1234") +
"&MERCH_NBR=" + HttpUtility.UrlEncode("1234567") +
"&DBA_NBR=" + HttpUtility.UrlEncode("1") +
"&TERMINAL_NBR=" + HttpUtility.UrlEncode("1") +
"&TRAN_TYPE=" + HttpUtility.UrlEncode("CCM1") +
"&AMOUNT=" + HttpUtility.UrlEncode("12.95") +
"&TRAN_NBR=" + HttpUtility.UrlEncode("1") +
"&BATCH_ID=" + HttpUtility.UrlEncode("5") +
"&CARD_ENT_METH=" + HttpUtility.UrlEncode("X") +
"&ACCOUNT_NBR=" + HttpUtility.UrlEncode("4111111111111111") + "&EXP_DATE="
+ HttpUtility.UrlEncode("2705");
```

```
byte[] data = encoding.GetBytes(postData);

// Prepare web request... HttpWebRequest myRequest =
(HttpWebRequest)WebRequest.Create("https://test.test.com");
myRequest.Method = "POST"; myRequest.ContentType="application/x-www-form-
urlencoded"; myRequest.ContentLength = data.Length;
Stream newStream=myRequest.GetRequestStream();
// Send the data. newStream.Write(data,0,data.Length); newStream.Close();

HttpWebResponse loWebResponse = (HttpWebResponse) myRequest.GetResponse();
Encoding enc = System.Text.Encoding.GetEncoding(1252); StreamReader
loResponseStream =
new StreamReader(loWebResponse.GetResponseStream(),enc);

string lcHtml = loResponseStream.ReadToEnd(); loWebResponse.Close();
loResponseStream.Close();

TextBox1.Text = lcHtml;

}
}
}
```

XML Secure Socket POST code example

PHP

```
<?php
// Get our request
$request = file_get_contents('credit_request.xml') or die('Could not read
file!');
// Open up the SSL socket to the host
$fp=fsockopen("ssl://test.test.com",6092,$errno,$errstr,30);
if(!$fp){
// We had an error connecting
echo "$errstr ($errno)\r\n";
return;
}else{
// We are connected
echo "Connected\n";
if (($bytesWritten = fwrite($fp, $request)) === false) {
// Error writing to the host
echo "ERROR: $errstr ($errno)\n";
return;
```

```
} else {
// Successful write
echo "SUCCESS: $bytesWritten bytes written\n";
}
}
// Get the response back from the host
if (($response = fgets($fp, 2048)) === false) {
echo "ERROR: $errstr ($errno)\n";
return;
} else {
// Print out the XML response
echo "SUCCESS: $response\n";
}
// Close our connection
fclose($fp);
?>
Contents of file "credit_request.xml":
<DETAIL CUST_NBR="1234" MERCH_NBR="1234567" DBA_NBR="1" TERMINAL_NBR="1">
<TRAN_TYPE>CCE1</TRAN_TYPE>
<AMOUNT>11.95</AMOUNT><ACCOUNT_NBR>4111111111111111</ACCOUNT_NBR>
<EXP_DATE>2703</EXP_DATE>
<CARD_ENT_METH>E</CARD_ENT_METH>
<INDUSTRY_TYPE>E</INDUSTRY_TYPE>
<BATCH_ID>1</BATCH_ID>
<TRAN_NBR>5</TRAN_NBR>
</DETAIL>
```