myob payby

JSON Web Service Interface

Technical Implementation User Guide

Version 1.0

CONTENTS

1. OVERVIEW

The JSON web service interface aims to:

+ Reduce the issues and overheads for merchants implementing multiple MYOB PayBy services.
+ Act as a facade so that a single interface can be used to access all MYOB PayBy services.
+ Allow merchants to easily change which MYOB PayBy services they use when their requirements change without having to engage in further implementation projects.
+ Reduce the complexity introduced by technologies such as XML, XSD and SSL certificates.

2. SUPPORTED SERVICES

Our goal is to expose all MYOB PayBy backend services via the JSON web service interface.

Supported services included in the JSON web service interface are:

+ CREDIT transaction processing, real-time and batch
+ CREDIT transaction reporting
+ DEBIT transaction processing, real-time and batch
+ DEBIT transaction reporting
+ Credit card tokenization and retrieval of card data via token
+ Some Pay Centre Web functionality
+ Fraud services

3. MESSAGE STRUCTURE

JSON

All messages are constructed using standard JavaScript Object Notation or JSON. The reason JSON is an ideal message format is that it is very simple and has a small number of data types.

All JSON messages are built on two structures:

+ A collection of name/value pairs
+ An ordered list of values.

JSON's basic types are:

+ Number – a signed decimal number that may contain a fractional part
+ String – a sequence of zero or more Unicode characters. Strings are delimited with double-quotation marks and support a backslash escaping syntax
+ Boolean – either of the values true or false
+ Array – an ordered list of zero or more values, each of which may be of any type. Arrays use square bracket notation with elements being comma-separated
+ Object – an unordered associative array (name/value pairs). Objects are delimited with curly brackets and use commas to separate each pair, while within each pair the colon ':' character separates the key or name from its value. All keys must be strings and should be distinct from each other within that object
+ null – An empty value, using the word null.

All JSON keys are case sensitive and use the same format of camel case with the first character in lowercase for example someKey, someOtherKey etc.

MESSAGE ENVELOPE

The JSON web service will support multiple backend MYOB PayBy services such as credit & debit real-time and batch transaction processing, Card Vault, Pay Centre Web and reporting so all requests and response messages will make use of the same message envelopes.

Request

```
{
 "service": {...service data...},
 "validateOnly": false,
"version": "1.5",
"requestDate": "2019-07-30 12:03:33"
}
```

The request message envelope is made up of the following name/value pairs:

+ service – object, required, contains the request data specific to the target service
+ validateOnly – boolean, required, if true a dummy response will be generated and the message will not be passed to the backend service layer. This is useful when testing the initial connectivity
+ requestDate – string, required, in the standard MYOB PayBy date format of yyyyMMdd HH:mm:ss

Response

```
{
 "service": {...service data...}
}
```

The response message envelope is made up of the following name/value pairs:

+ service – an object type, contains the response data specific to the target service.

All dates must be string values in the standard MYOB PayBy date format of yyyy-MM-dd HH:mm:ss for date-time values and yyyy-MM-dd for date-only values.

4. JSON MESSAGE SCHEMA

Schema support for JSON is not extensive at this stage but the schema can still be used to generate code and also serves as a form of documentation. Please see http://json2csharp.com/ and http://www.jsonschema2pojo.org/ for examples of utilities that can generate code from JSON schemas and JSON sample messages.

Any new services developed at MYOB PayBy in future may also be added to the JSON interface.

For this reason it is more practical to provide schemas and reference documentation separately for each supported service.

+ Request schema: link used for Java utility
+ Response schema: link used for Java utility
+ Request JSON: link used for C# utility
+ Response JSON: link used for C# utility

MARSHALLING FRAMEWORKS

Producing JSON messages manually and processing response messages manually is fairly easy to do but, in a real application, marshalling frameworks are typically used to convert classes to JSON and JSON back to classes.

Populating objects in code with data and then converting them to JSON via marshalling is less likely to produce errors in the JSON format and unknown fields or fields with incorrect names etc.

For Java there are a number of good JSON marshalling frameworks such as:

+ Jackson
+ GSON

For .NET there are also a few but we have only used one of them:

+ Json.NET

5. BASIC CONNECTIVITY

HTTP POST REQUEST

The JSON web service only accepts POST requests and also requires the AUTHTOKEN header. An auth token will be provided to you by MYOB PayBy support along with a client ID to use when processing requests.

To get basic connectivity working you will need:

+ An authentication token – this is a GUID unique to each MYOB PayBy client, for example c12f9140-8574-4c66-8dce-79333f24e1c1
+ A clientId – this is an integer 8 digits long for example 19000237

During development you will need to post requests to the MYOB PayBy TEST environment at the following endpoint: https://test-merchants.paycorp.com.au/paycorpwebservice/InterfaceServlet

TESTING WITH CURL

The first thing to verify is that you can connect to the MYOB PayBy JSON web service using dummy data. As mentioned in the message envelope section if you set validateOnly to true the request is not passed to the MYOB PayBy backend services and a simple dummy response will be returned. This makes it easy to get the basic connectivity working without the need to worry about passing in valid service request data.

CURL is another useful tool that can be used to test connectivity without having to write any code. You will need to install CURL if you have not done so already. On Ubuntu and other Linux operating systems this is as easy as running the command sudo apt-get install curl. For help installing CURL please see here: https://curl.haxx.se/download.html

For Windows users another option is to use WGET or GetWebFile via Powershell or skip this test and move on to testing with the example code as there is a C# example for .NET.

Once CURL has been installed try running the following command at the terminal to test connectivity:

```
curl X
POST H
'Content-type:application/json' -H
'AUTHTOKEN: fb0e0f5a-3931-4c15-bac4-8c10b4da9999'–data'
{
        "api":"JAVA",
        "version":"1.5",
        "msgId":"469c8a04-d0e3-40a2-b58f-6c9532d1e441",
        "operation":"PAYMENT_REAL_TIME",
        "requestDate":"2019-07-30T12:03:33.576+1000",
        "validateOnly":false,
        "requestData":{
                "clientId":10022000,
                "creditCard":{
                        "holderName":"Automation",
                        "number":"4564456445644564",
                        "expiry":"1222",
                        "secureId":"123"
                },
        "transactionType":"PURCHASE",
        "transactionAmount":{
                "totalAmount":0,
                "paymentAmount":1299,
                "serviceFeeAmount":0,
                "withholdingAmount":0,
                "currency":"AUD"
        },
        "clientRef":"null"
  }
}

 'https://test-merchants.paycorp.com.au/rest/service/proxy'
```

If all goes well you should see a response message similar to this:


```
{
   "msgId": "469c8a04-d0e3-40a2-b58f-6c9532d1e441",
   "sessionStatus": null,
   "error": null,
   "responseData": {
      "txnReference": "9008100009862604",
      "responseCode": "00",
      "responseText": "APPROVED",
      "settlementDate": "2019-07-30",
      "authCode": "470581"
   }
}
```


TESTING WITH POSTMAN

Another very useful testing utility is POSTMAN but it has the limitation that it only works in the Google Chrome web browser but installing Chrome is well worth the effort even if it is just to use POSTMAN. Chrome is easy to install as is POSTMAN.

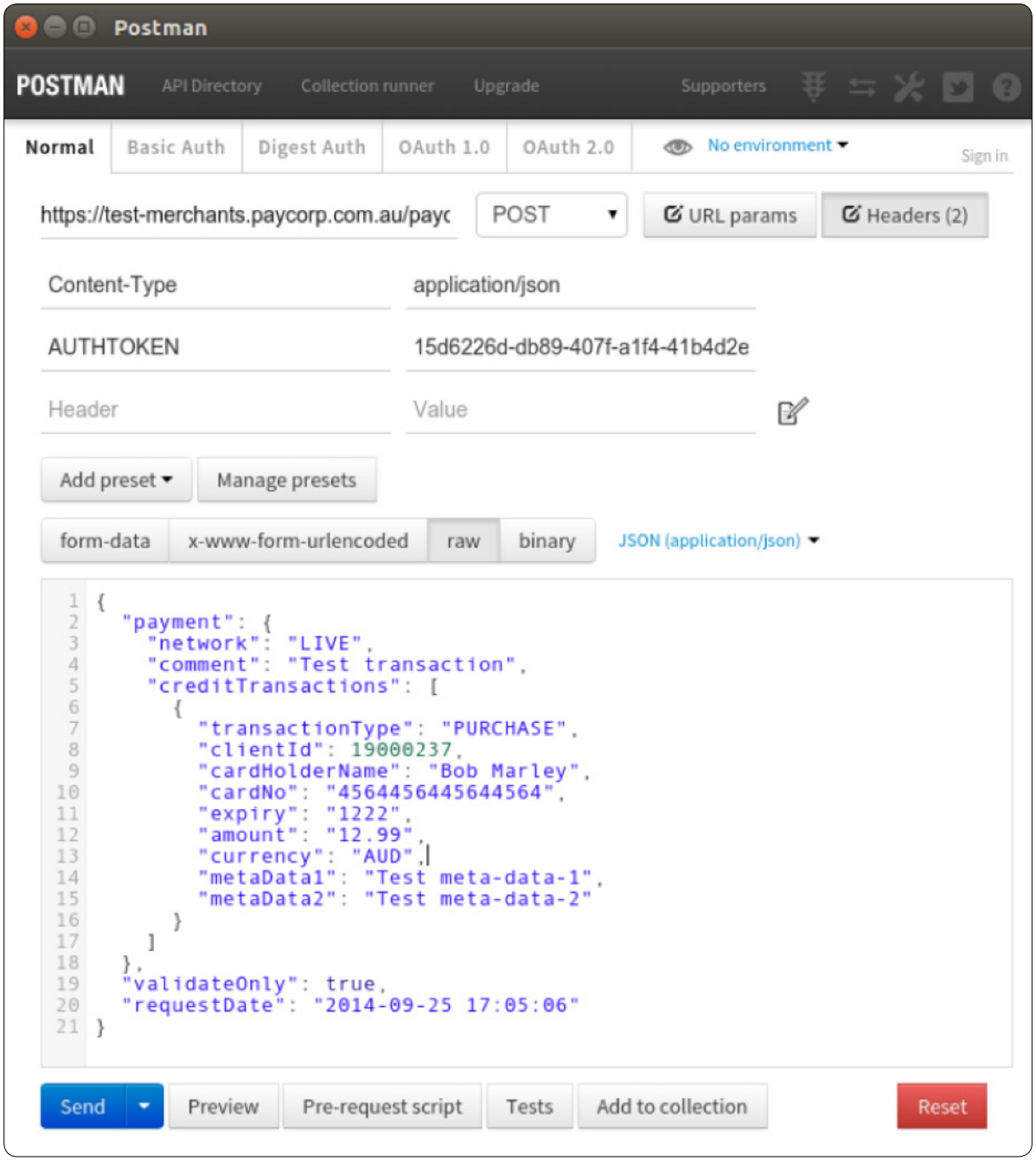
Once you have Chrome and POSTMAN installed you can access POSTMAN in Chrome by clicking on the app launcher icon. 

It is located top left when running on Ubuntu for example. Then click the POSTMAN button. 

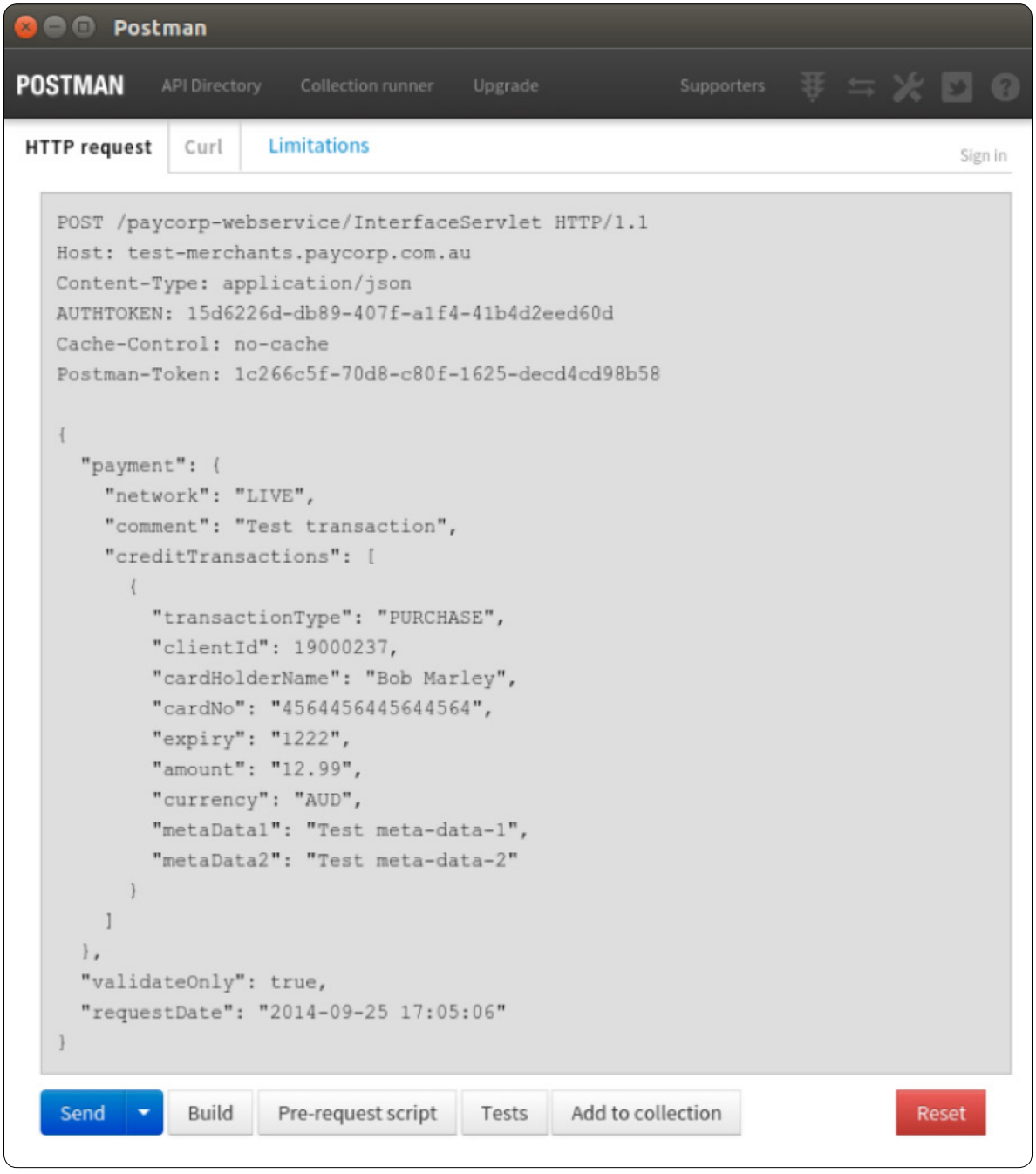
Once you have POSTMAN opened in a Chrome tab there are a few things to setup before you can begin testing the MYOB PayBy JSON web service.

1. Select the Normal tab in the main top menu. It should be selected by default.
2. Select the POST web method at the top right of the screen. The default will be GET.
3. Select the raw data format. The default will be form-data.
4. Select the JSON (application/json) content type. The default will be Text.
5. Click the Headers button at the top right of the screen.
6. Add a new AUTHTOKEN header with the value being the authentication token GUID you were provided with or the test GUID: fb0e0f5a-3931-4c15-bac4-8c10b4da9999.
7. Copy and paste the JSON request message to test into the text area.
8. Enter the URL of the TEST MYOB PayBy JSON web service: https://test-merchants.paycorp.com.au/paycorpwebservice/InterfaceServlet

The completed setup is shown below.



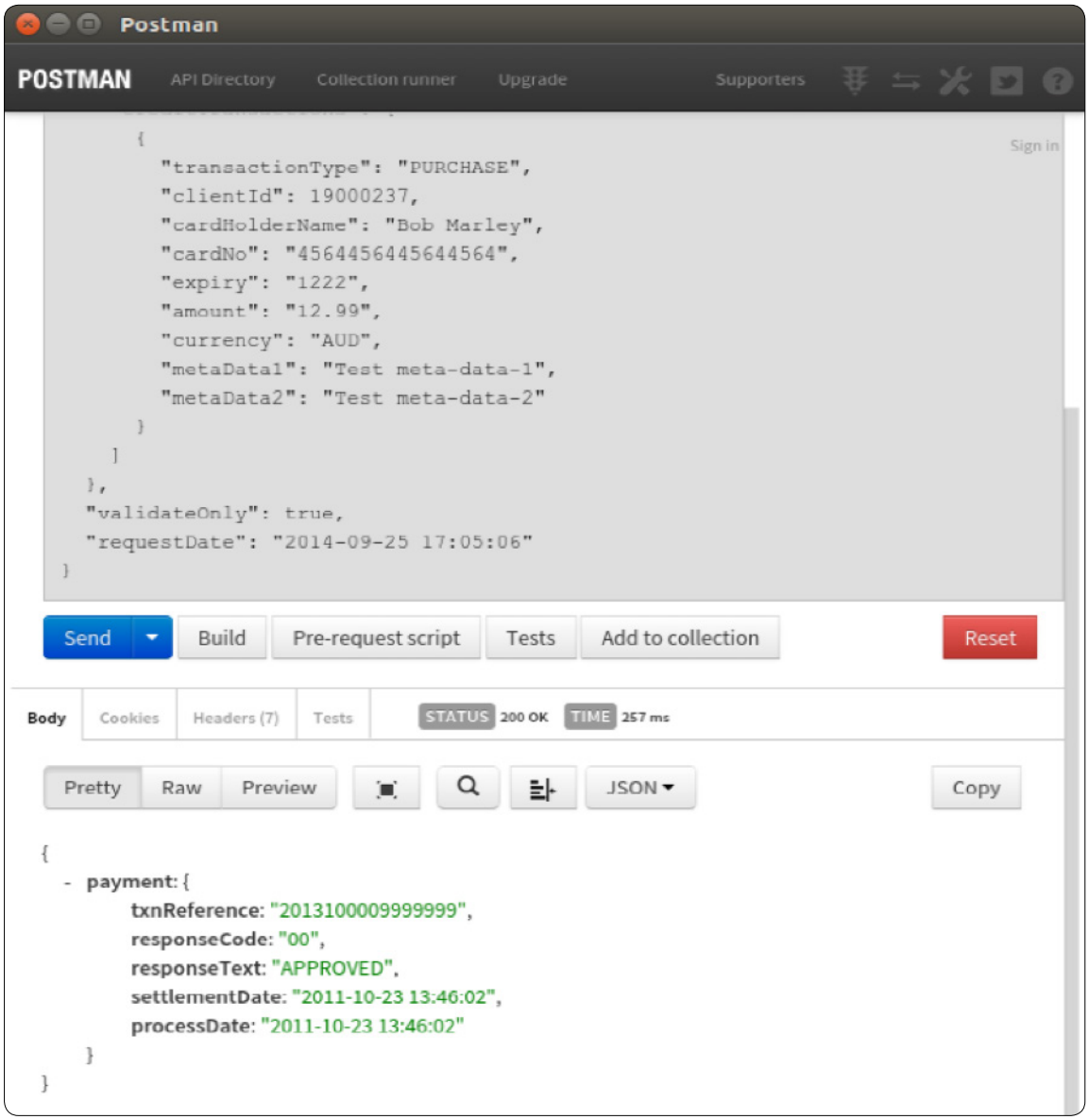
You can PREVIEW the actual HTTP POST request that will be sent. It should look something like this:



When you are ready you can then click the send button. The response should be displayed as follows:

When you are ready, you can click the Send button.

The response shoud be as follows:



6. SECURITY CONSIDERATIONS

Security is always critical when working with sensitive information such as credit card numbers and we would advise all merchants to try and implement all of the added security features offered by the MYOB PayBy JSON web service interface.

All MYOB PayBy web service interfaces use an HTTPS connection so that all data transmitted between merchants and MYOB PayBy is encrypted. Although using HTTPS & SSL is much more secure than using standard HTTP, SSL is still vulnerable to certain attacks and as a result the JSON web service interface has a few added security features to help improve security.

At present there are only 2 added security features available:

1. The use of an authentication token to validate user identity – WEAK but MANDATORY.
2. The use of a unique HMAC per request based on a shared secret – STRONG but OPTIONAL.

Because some merchants may not possess all of the required technical skills to implement the HMAC security, in some cases this feature may not be required. It is also possible to disable the HMAC security during initial development so that if merchants are finding it difficult to get the HMAC security working this will not hold up the rest of the implementation project.

AUTHENTICATION TOKEN

To identity merchants who use the MYOB PayBy JSON web service, merchants will be issued a unique authentication token. The auth token is a random string of characters called a GUID. We use a GUID rather than a simple username because it is more difficult for a malicious user to guess, copy or remember. An example of such a GUID is fb0e0f5a-3931-4c15-bac4-8c10b4da9999.

The auth token GUID needs to be supplied with all requests to the JSON web service in the form of an HTTP header for example:

```
POST /rest/service/proxy
HOST: test-merchants.paycorp.com.au
CONTENT-TYPE: application/json
AUTHTOKEN: fb0e0f5a-3931-4c15-bac4-8c10b4da9999
```

HMAC SECURITY

HMAC stands for Hashbased message authentication code and it is a way to verify both the data integrity and the authentication of a message. A HASH is an algorithm that can be applied to a message and the result of the HASH algorithm is a unique string of data.

Before sending a request message to MYOB PayBy the JSON message data can be hashed to produce a value that is unique to the message. Even if the message data changes in a very small way the hash value produced would be completely different. This means any changes in the message content can be detected by comparing the HASH values.

An example of a hash is shown below:

```
Message : "The quick brown fox jumps over the lazy dog"
HASH : 0xf7bc83f430538424b13298e6aa6fb143ef4d59a14946175997479dbc2d1a3cd8
```

In a similar way the HASH of the following JSON message:

```
{
 "payment": {
  "network": "LIVE",
  "comment": "Test transaction",
  "creditTransactions": [
   {
    "transactionType": "PURCHASE",
    "clientId": 19000237,
    "cardHolderName": "Bob Marley",
    "cardNo": "4564456445644564",
    "expiry": "1222",
    "amount": "12.99",
   }
  ]
 },
 "validateOnly": true,
```

```
 "requestDate": "2014-09-25 17:05:06"
}
```

Produces the following value:

HASH : 260283c43b06808bd70036fbdb49003ca346a15f493c97333cd0cd334d1bf989

Another property of the HASH algorithm is that it can be repeated and will always produce the same output string when applied to a given message input. So when a request message arrives at MYOB PayBy servers that includes a HASH value the message content can be hashed on the MYOB PayBy side and the MYOB PayBy generated hash value can be compared with the merchant generated hash value sent along with the request.

If the MYOB PayBy and merchant hash values match then we can be sure that the request arrived at MYOB PayBy in the exact same state it was in when it left the merchant servers. This way we can verify that the message content has not been distorted and was not tampered with while in transit over the internet.

Added to that, the HASH algorithm can be combined with a secret.

Hashing the same message content using 2 different secrets will produce 2 different results for example:

Message : "The quick brown fox jumps over the lazy dog"
Secret : "key"
HASH : f7bc83f430538424b13298e6aa6fb143ef4d59a14946175997479dbc2d1a3cd8
Message : "The quick brown fox jumps over the lazy dog"
Secret : "otherkey"
HASH : adea30df7e096340a0532da97d7cd62919cbfb41075d3597fd61b78f679c2a40

As you can see using 2 different secrets and the same message contents produces 2 different results. This shows how an HMAC is the combination of a HASH algorithm and a secret key. This means that we can also use the HMAC to verify that the merchant used the correct secret to produce the hash value.

The main advantage of this is that although the HMAC value is transmitted over the internet, even if someone managed to intercept and decrypt the data sent over the HTTPS connection they would not be able to reproduce the HMAC without the secret. The secret itself is never transmitted over the internet and is kept safely on the merchant servers and on MYOB PayBy servers.

Because producing an HMAC is a common requirement most programming languages will have the required algorithms included.

To send the generated HMAC to MYOB PayBy include it as an HTTP header.

HMAC: f7bc83f430538424b13298e6aa6fb143ef4d59a14946175997479dbc2d1a3cd8

7. APPENDIX A – EXAMPLE MESSAGES

Basic test message, returns a DUMMY response


```
{
 "payment": {
  "network": "LIVE",
  "comment": "Test transaction",
  "creditTransactions": [
   {
    "transactionType": "PURCHASE",
    "clientId": 19000237,
    "cardHolderName": "Bob Marley",
    "cardNo": "4564456445644564",
    "expiry": "1222",
    "amount": "12.99",
    "currency": "AUD",
    "metaData1": "Test meta-data-1",
    "metaData2": "Test meta-data-2"
   }
  ]
 },
 "validateOnly": true,
 "requestDate": "2014-09-25 17:05:06"
}
```


Expected response


```
{
 "payment": {
  "txnReference": "2013100009999999",
  "responseCode": "00",
  "responseText": "APPROVED",
  "settlementDate": "2011-10-23 13:46:02",
  "processDate": "2011-10-23 13:46:02"
 }
}
```


Basic report request


```
{
 "report": {
  "network": "LIVE",
  "reportType": "CREDIT_CARD",
  "batchId": 498818
 },
 "validateOnly": false,
 "msgProcessId": "7e58467b-22d6-4264-adc7-5d161ee66843",
 "requestDate": "2014-09-26 17:06:26"
}
```

Expected response


```
{
 "report": {
  "generatedDate": "2014-09-26 17:12:31",
  "network": "LIVE",
  "creditTransactions": [
   {
    "type": "PURCHASE",
    "clientId": 10004329,
    "cardHolderName": "Jo Blohh",
    "cardNo": "456445xxxxxx4564",
    "expiry": "1020",
    "amount": "1.00",
    "currency": "AUD",
    "terminalType": 0,
    "metaData1": "",
    "metaData2": "8f711421-f49f-4a6a-bbe5-903893c9773f",
    "threeDSecureCAVV": "",
    "cardType": "VISA",
    "responseCode": "00",
    "responseText": "APPROVED (TEST TRANSACTION ONLY)",
    "stan": "750916",
    "settlementDate": "2014-09-10 00:00:00",
    "authCode": "026106",
    "merchantId": "",
    "terminalId": 0,
    "secureIdSupplied": false,
    "txnReference": "1008100006580294",
    "eventTime": "2014-09-10 14:00:23",
    "batchId": 498818
   }
  ],
  "debitTransactions": []
 }
}
```

8. APPENDIX B – HMAC CODE EXAMPLES

Java example for generating an HMAC, where secret will be provided and will be something like HelloWorld1234, the data is the JSON message.

```
public class HmacUtils {

 public String generateMac(final String secret, final String data)
  throws java.security.NoSuchAlgorithmException,
  java.security.InvalidKeyException,
  java.io.UnsupportedEncodingException,
  org.apache.commons.codec.DecoderException {

  byte[] b = secret.getBytes("ISO88591");
  javax.crypto.SecretKey key =
  new javax.crypto.spec.SecretKeySpec(b, "HmacSHA256");
  javax.crypto.Mac m = javax.crypto.Mac.getInstance("HmacSHA256");
  m.init(key);
  m.update(data.getBytes("ISO88591"));
  return new

String(org.apache.commons.codec.binary.Hex.encodeHex(m.doFinal()));
 }
}
```

Same example in C#

```
public class CryptoUtil
{
 public string CreateHmacToken (string json, string secret)
 {
  secret = secret ?? "";
  var encoding = new System.Text.ASCIIEncoding ();
  byte[] keyByte = encoding.GetBytes (secret);
  byte[] messageBytes = encoding.GetBytes (json);
  using (var hmacsha256 = new HMACSHA256(keyByte)) {
  byte[] hashmessage = hmacsha256.ComputeHash
(messageBytes);
  String hmac = Convert.ToBase64String (hashmessage);
  string hex = BitConverter.ToString(hashmessage)
  .Replace("",
  string.Empty);
  Console.WriteLine (hmac);
  Console.WriteLine (hex);
  return hex.ToLower();
  }
 }
}
```

9. APPENDIX C – JSON MARSHALLING CODE EXAMPLES

A Java example using the Gson marshalling framework.

First you need a custom serializer to specify how dates will be handled.

```
public class DateSerizlier implements JsonDeserializer<Date>,
JsonSerializer<Date> {

 private static final String[] DATE_FORMATS = new String[]{
  "yyyyMMdd
  HH:mm:ss",
  "yyyyMMdd"
 };

 @Override
 public Date deserialize(JsonElement jsonElement, Type typeOF,
  JsonDeserializationContext context) throws JsonParseException {
 return parseDate(jsonElement.getAsString());
 }

 protected Date parseDate(String date) {
 for (String format : DATE_FORMATS) {
 try {
  return new SimpleDateFormat(format).parse(date);
  } catch (ParseException e) {
  }
 }
 throw new JsonSyntaxException("Unparseable date: \"" + date
  + "\". Supported formats: " +
 Arrays.toString(DATE_FORMATS));
}

 @Override
 public JsonElement serialize(Date date, Type type,
 JsonSerializationContext jsc) {
 return new JsonPrimitive(
  new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(date));
 }
 }
```

Now a class to handle the marshalling

```
public class JsonUtils {
 private Gson getGson() {
  GsonBuilder gsonbuilder = new GsonBuilder();
  gsonbuilder.registerTypeAdapter(Date.class, new DateSerizlier());
  gsonbuilder.setPrettyPrinting();
  return gsonbuilder.create();
 }

 public String toJson(PaycorpRequest paycorpRequest) throws
  JsonMappingException, JsonProcessingException {
  return getGson().toJson(paycorpRequest);
 }
}
```

A C# example using the Json.NET marshalling framework.

You can also create a custom serializer to specify how dates will be handled using Json.NET.

```
public class JsonUtil
{
 public string getJson (PaycorpRequest request)
 {
   /*
   Now serialize to JSON, a few things to note:
   1. We want the enum name not its INT value.
   2. If a property is NULL it should not be included in the JSON
   3. JSON field names start with lower case.
   4. Formatting is optional.
  */
  JsonSerializer serializer = new JsonSerializer ();
  serializer.Converters.Add (new StringEnumConverter ());
  serializer.NullValueHandling = NullValueHandling.Ignore;
  //Makes unit test assert complex
  //serializer.Formatting = Formatting.Indented;
  String responseString = "";
  using (StringWriter sw = new StringWriter())
  using (JsonWriter writer = new JsonTextWriter(sw)) {
  serializer.Serialize (writer, request);
  //Console.Write (sw.ToString ());
  responseString = sw.ToString ();
  }
  return responseString;
 }
}
```

10. APPENDIX D – HTTPS POST CODE EXAMPLE

C# example for posting the request via HTTPS

```
public class HttpsPostUtil
{
 public string HttpPost (string json, string hmacToken,
 string authToken)
 {
  string url =
  "https://test-merchants.paycorp.com.au/paycorp-webservice/InterfaceServlet"
  ;
  HttpWebRequest req = WebRequest.Create (new Uri (url)) as
  HttpWebRequest;
  req.Method = "POST";
  req.ContentType = "application/json";
  //Convert JSON into bytes
  byte[] byteData = UTF8Encoding.UTF8.GetBytes (json);
  req.ContentLength = byteData.Length;
  req.Headers.Add ("AUTHTOKEN", authToken);
  req.Headers.Add ("HMAC", hmacToken);
  ServicePointManager.ServerCertificateValidationCallback = new
  System.Net.Security.RemoteCertificateValidationCallback
  (AcceptAllCertifications);
  // Send the request:
  using (Stream post = req.GetRequestStream()) {
  post.Write (byteData, 0, byteData.Length);
 }
  // Pick up the response:
  string result = null;
  using (HttpWebResponse resp = req.GetResponse() as HttpWebResponse) {
  StreamReader reader = new StreamReader
 (resp.GetResponseStream ());
  result = reader.ReadToEnd ();
 }
return result;

 }
 public bool AcceptAllCertifications (
 object sender,
 X509Certificate certificate,
 X509Chain chain,
 SslPolicyErrors sslPolicyErrors)
 {
 return true;
 }
}
```

11. TECHNICAL SUPPORT HELP DESK

PAYBY SUPPORT

Email: techsupport@myobpayby.com
Phone: 1300 303 742

Available Monday to Friday from 9:00am-5:30pm AEST.

myob payby
the way to pay