Guide to Integration Without a DataCash API

Version: 1.0
Publication Date: 19-May-2009

Contents

A. Introduction

The DataCash Payment Gateway (DPG) is a service that accepts transaction requests in the form of XML documents submitted via HTTPS POST operations. Similarly, the result of each transaction is returned to the calling process as an XML document in the response of the POST operation.

For convenience, DataCash have written, and provide limited support for, a number of client API libraries that aim to simplify integration with the DPG using a range of programming languages and platforms. Whilst the full functionality and use of these APIs varies slightly from one language to another, the main functions of each one are routines to simplify the creation and parsing of DataCash XML request and response documents, along with code to perform the required HTTPS POST request/response exchange.

Although these APIs generally provide relatively quick and easy means of integrating with the DPG, there are a few situations where it might be preferable to implement these functions locally, without using an API. DataCash are quite happy for people to do so; this document aims to give a brief introduction to some of the key areas that would form the basis of such a local implementation.

Language-specific guidance and examples will be provided for four languages:

- Microsoft .NET (specifically C#, though other .NET languages will be similar)
- Java
- Perl
- PHP

Every attempt has been made to ensure that code examples used in this document are valid and correct at time of writing. However, they are intended to be used as guides only, to be used as starting points for production-worthy integrations; furthermore libraries and technologies are liable to change over time.

The guide assumes a basic understanding of the HTTP(S) protocol and the XML document format.

Note that the purpose of this document is solely to cover message exchange between your systems and the DataCash Payment Gateway. We offer no support in the writing or maintaining of shopping-cart or other order-management systems.

B. Language-independent Information

B.1. HTTPS POST

HTTP POST requests differ from GET requests in that instead of requesting a resource based solely on the URL in the request, a POST request normally contains an additional block of data. Traditionally they were used to transmit the contents of complex web forms (as filled in by a human user) when requesting a dynamically generated page from a web server; the form and the resulting web-page (returned in the HTTP response) would have been rendered by an HTML browser.

Lately it has become increasingly commonplace to see HTTP POSTs being used by applications independently of web browsers, in which the request and responses are documents designed to allow applications to interoperate remotely with each other. In particular, due to its platform neutrality, XML has been the most common document format for both request and response.

This is essentially how the DPG works. When a client wishes to invoke a service provided by the Gateway (for example, to conduct a single bankcard transaction), their application must make an HTTPS (HTTP Secure) POST request to the DPG's URL (this URL will be supplied in your welcome pack), in which the additional data is an XML document containing details of the desired transaction. Instead of an HTML web page in response, the DPG will return a similar XML document containing details of the result of the transaction.

Example of an HTTP POST request

```
POST /Transaction HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 452
Connection: Keep-Alive
Host: localhost:8808

<?xml version="1.0" encoding="UTF-8"?><Request><UserAgent><Libraries
bundle="Microsoft .NET" version="1.1.3.4"
/></UserAgent><Authentication><client>99002000</client><password>****
</password></Authentication><Transaction><CardTxn><Card><pan>44443333
22221111</pan><expirydate>10/10</expirydate></Card><method>pre</metho
d></CardTxn><TxnDetails><amount>100</amount><merchantreference>120220
09102317</merchantreference></TxnDetails></Transaction></Request>
```

This is an actual HTTP POST as submitted by the DataCash .NET API to a test gateway instance. There are a couple of things worth noting here. Firstly, the content-type must always be set to the MIME type `application/x-www-form-urlencoded`.

Also, in the above request, we have used the .NET default connection type `Keep-Alive`. This instructs the DataCash web server to maintain the underlying connection upon completion of a single exchange, and to reuse it for subsequent exchanges. We have had a small number of reports of issues with this setting on some platforms. If concerned, this is something that most HTTPS libraries will allow you to disable.

The Payment Gateway response might look something like the below:

Example of an HTTP POST response

```
HTTP/1.1 200 OK
Date: Thu, 12 Feb 2009 10:23:20 GMT
Server: Apache
Connection: close
Content-Type: text/plain; charset=iso-8859-1

<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <CardTxn>
    <authcode>190617</authcode>
    <card_scheme>VISA</card_scheme>
    <country>United Kingdom</country>
  </CardTxn>
  <datacash_reference>4600900012345673</datacash_reference>
  <merchantreference>12022009102317</merchantreference>
  <mode>TEST</mode>
  <reason>ACCEPTED</reason>
  <status>1</status>
  <time>1234434201</time>
</Response>
```

Note that depending on the HTTP protocol version the actual response from the Gateway may look significantly different; perhaps taking advantage of techniques such as chunked transfer coding or use of the CONTINUE mechanism. In the above example, HTTP 1.0 was specified in the request purely to ensure that the example response was as simple as possible. Furthermore, in order to capture the actual HTTP request and response, this exchange was performed over plain HTTP. For security, real transaction processing is restricted to the HTTPS protocol (specifically using SSL version 3 with 128 bit encryption or better). Because of subtleties such as these, it is *highly recommended* that a stable and mature HTTPS protocol library is used for all message exchanges.

In order to provide high-availability, the actual IP address that the DataCash Payment Gateway URL resolves to in DNS may change without notice (although we do publish a “pool” of possible addresses, and will keep you updated should this pool change). It is therefore strongly advised that the URL you submit requests to make use of the DNS name provided, and that any DNS caching honours a low “time to live” in order that such changes are picked up in a timely manner. This is the default behaviour for most platforms, though there is at least one case (documented in the language-specific notes below) where it needs to be set explicitly. It should further be noted that should your network infrastructure requires outgoing network requests be explicitly sanctioned by some sort of firewall, you will need to ensure that all IP addresses in the aforementioned address pool be included in your firewall configuration.

B.2. XML Request/Response

For details of the DataCash XML request and response formats, please see the Developers’ Guide document, available from the DataCash Developers' Area.

Although an in-depth description of the XML document format is beyond the scope of this document, it is worth mentioning encoding issues here. Historically it was generally the case that strings were represented using one byte per character. However, the need for accented characters or non-Latin alphabets means that this is no longer sufficient. Unfortunately it is now the case that different platforms adopt different default underlying representations of strings. This clearly makes interoperability between platforms more complex, for example when converting a string into a sequence of bytes for transmission on a network (referred to as the “wire-representation” of the string).

To resolve this problem, an XML document may be prepended with a “declaration”, allowing one to explicitly state the character encoding that is being used to represent the document (XML parsers are required to be able to read and honour an XML declaration that is itself encoded using almost any standard encoding). By ensuring that the encoding specified in your XML declaration matches whatever library method is being used to convert your XML document to its wire-representation, your document will be correctly read and understood by the DataCash Payment Gateway. An example XML declaration, specifying the `UTF-8` encoding is shown below:

Example XML declaration

```
<?xml version="1.0" encoding="UTF-8"?>
```

C. Language-specific Information

We will now present some example code and guidance that can be used to perform DataCash transactions without using a DataCash API in a number of popular programming languages. The examples should not be considered complete and ready for production, but should be sufficient to get you started. Also, it should be noted that you are not restricted to the languages mentioned, nor do you have to use the particular libraries discussed here. There are usually several XML and HTTP libraries available for use, each with their own pros and cons in terms of, for example, ease of use and/or scalability.

Note that we have omitted any logging code from the specific examples below. You will most likely want to write information to some sort of log file during transaction processing. However, it is important that any logging should not write sensitive information to the file in plaintext. Specifically, DataCash (or other) passwords and card details must be obfuscated before logging. CV2 numbers in particular must *never* be stored beyond the lifespan of a single transaction.

C.1. C#/Microsoft .NET

Although we will restrict our code examples in this section to the C# language, the nature of the .NET framework means that code written in any language that can compile down to CIL bytecode. For example, VB.NET code should be fairly straightforward to deduce based on the C# examples given. The examples are largely based on the DataCash .NET API which was written for the .NET Framework version 1.1, though we have no reason to believe that they will not compile and work with any version of the Framework.

C.1.1. XML Generation/Parsing

The first thing your implementation will need to do is create an XML document containing details of the desired transaction. While there are several XML processing libraries available, the most obvious starting point would be the XML functionality contained within the .NET framework, in the `System.XML` namespace. This is based around the `XmlDocument` class, and allows hierarchical navigation of XML documents for both creating and parsing the XML. It is extremely intuitive, and well documented, so there is little point going into details here, save to provide small examples of both building and parsing DataCash requests and responses:

Request generation example

```
private XmlDocument createRequest()
{
      // Create the document, and add an XML declaration
      XmlDocument doc = new XmlDocument();
      XmlDeclaration dec =
            doc.CreateXmlDeclaration("1.0", "utf-8", null);
      doc.AppendChild(dec);

      // Define the root (top-level) element
      XmlElement root = doc.CreateElement("Request");
      doc.AppendChild(root);

      // Add some new container elements
      XmlElement transactionElm = doc.CreateElement("Transaction");
      root.AppendChild(transactionElm);
      XmlElement txnDetailsElm = doc.CreateElement("TxnDetails");
      transactionElm.AppendChild(txnDetailsElm);

      // Add a "leaf" element with string value
      XmlElement merchantRefElm =
            doc.CreateElement("merchantreference");
      XmlText mref = doc.CreateTextNode("12345601");
```

```
        merchantRefElm.AppendChild(mref);
        txnDetailsElm.AppendChild(merchantRefElm);

        // Add a "leaf" element with value and single attribute
        XmlElement amountElm = doc.CreateElement("amount");
        amountElm.SetAttribute("currency", "GBP");
        XmlText amt = doc.CreateTextNode("100.0");
        amountElm.AppendChild(amt);
        txnDetailsElm.AppendChild(amountElm);

        return doc;
}
```

This code results in the following XML document:

Request generation example results

```
<?xml version="1.0" encoding="utf-8"?>
<Request>
  <Transaction>
    <TxnDetails>
      <merchantreference>12345601</merchantreference>
      <amount currency="GBP">100.0</amount>
    </TxnDetails>
  </Transaction>
</Request>
```

Parsing a response document is similarly straightforward:

Response parsing example

```
private string getDcReference(string strResponse)
{
        // Parse the XML from a string
        XmlDocument doc = new XmlDocument();
        doc.LoadXml(strResponse);

        // Get the root node
        XmlElement root = doc.DocumentElement;

        // Find all descendants with the desired name
        XmlNodeList nodes =
              root.GetElementsByTagName("datacash_reference");

        // Assume only one "hit" and return the contained text
        return nodes[0].InnerText;
}
```

Naturally you will want to include more error-checking than in the above example, but it should give a reasonable indication of how simple it is to extract information from a response.

C.1.2. HTTPS Submission

Once you have built your XML request document you will need to submit it to the DataCash Payment Gateway via HTTPS POST. Again, the most obvious starting point is to use the .NET Framework's built-in library to do this; this time the key class is `HttpWebRequest` in the `System.Net` namespace. The first thing to do is create an instance of this class:

Creating the WebRequest

```
string host = "https://testserver.datacash.com/Transaction";
HttpWebRequest httpWebRequest = null;

try
{
	httpWebRequest = (HttpWebRequest)WebRequest.Create( host );
}
catch (UriFormatException e)
{
	// Couldn't parse URL
	return;
}
```

A number of properties must then be set on the request object:

Setting request properties

```
httpWebRequest.Method = "POST";
httpWebRequest.ContentType = "application/x-www-form-urlencoded";

// int timeoutSeconds is your desired timeout value in seconds
httpWebRequest.Timeout = timeoutSeconds * 1000;

// Content length in bytes
UTF8Encoding encoding = new UTF8Encoding();
string xmlString = xmlDoc.OuterXml;
byte[] xmlBytes = encoding.GetBytes(xmlString);
httpWebRequest.ContentLength = xmlBytes.Length;
```

Note that a common error is to use the string length of the XML document as the content length. This will work as long as you restrict the character set of the request to single-byte characters. However, should the request contain characters that will be encoded as multiple bytes, this will yield an incorrect content length. The actual encoding you choose to use is largely immaterial, *as long as the XML declaration matches the wire-representation*. Thus in our examples, which have used the declaration `<?xml version="1.0" encoding="utf-8"?>`, we must ensure that we use an instance of the `UTF8Encoding` class to produce the wire-representation.

Some optional properties may be required, depending on your network infrastructure and server platforms. In particular, the `Proxy`, `KeepAlive` and `ProtocolVersion` properties may be needed. Setting a proxy will override the default proxy configured on your server with a specific one. This proxy may be simply defined by a proxy URL, or more complex constructors allow you to specify credentials for an authenticating proxy.

With `KeepAlive` set to its default value of `true`, the Framework will attempt to maintain the underlying socket connection once the request has completed, and re-use it for any subsequent requests to the same endpoint. We have received reports of connection issues from a small number of customers that seem to be related to this property. If experiencing intermittent connection errors (or simply want to pre-emptively avoid such errors) you may wish to set this to `false`; it is also normally worth setting the `ProtocolVersion` property at the same time to force your request to specify HTTP 1.0 rather than 1.1:

Setting optional properties

```
// Setting a simple proxy
string proxyAddress = "http://proxyserv:8080";
try
{
	httpWebRequest.Proxy = new WebProxy(proxyAddress);
}
catch (UriFormatException e)
```

```
{
      // Couldn't parse URL
      return;
}

// Disable KeepAlive
httpWebRequest.KeepAlive = false;
httpWebRequest.ProtocolVersion = HttpVersion.Version10;
```

Finally, the content is added, the request sent and the response read:

Performing the request and reading the response

```
Stream str = null;
try
{
      str = httpWebRequest.GetRequestStream();
      str.Write( xmlBytes, 0, xmlBytes.Length );
      str.Close();
}
catch (Exception ex)  // Or catch specific exceptions
{
      // Handle exception
      return;
}

HttpWebResponse httpWebResponse =
        (HttpWebResponse)httpWebRequest.GetResponse();

string result = "";
using (StreamReader streamReader =
      new StreamReader(httpWebResponse.GetResponseStream()))
{
      try
      {
            result = streamReader.ReadToEnd();
            streamReader.Close();
      }
      catch (Exception e)
      {
      // Handle exception
            return;
      }
}

// We get the HTTP Headers to start with, so
// knock them off before creating the XML Document
int startOfXML = result.IndexOf( "<?xml" );
string xmlResponse = result.Substring( startOfXML );

// Now load the string into an XmlDocument
XmlDocument responseDocument = new XmlDocument();
responseDocument.LoadXml( xmlResponse );
```

Again note that the sample code here is missing several key features, such as logging, proper error-handling etc. You might also consider the use of a separate thread for the actual remittance to ensure that timeouts are guaranteed to be honoured and to protect the main running thread of your application.

C.2. Java

Sample code here is largely based on the DataCash Java API, which was written for Java version 1.4.2. To the best of our knowledge they should work with newer versions of Java, though some areas might be improved by taking advantage of newer language features.

C.2.1. XML Generation/Parsing

Several XML libraries are available for Java platforms; the one that we find most suitable for our purposes is JDOM, available from http://www.jdom.org. The following is based on JDOM version 1.0, though again we have no reason to believe that it will not work with other release versions of the library.

Again, JDOM is extremely intuitive to use, so some simple code will suffice as an example:

Example XML generation

```
import org.jdom.*;

...

private Document createDocument() {
        // Create the document
        Document doc = new Document();

        // Add a root element
        Element root = new Element("Request");
        doc.setRootElement(root);

        // Add some container elements
        Element transactionElm = new Element("Transaction");
        root.addContent(transactionElm);
        Element txnDetailsElm = new Element("TxnDetails");
        transactionElm.addContent(txnDetailsElm);

        // Add a simple leaf node
        Element merchantRefElm = new Element("merchantreference");
        merchantRefElm.setText("12345601");
        txnDetailsElm.addContent(merchantRefElm);

        // Add a leaf node with single attribute
        Element amountElm = new Element("amount");
        amountElm.setAttribute("currency", "GBP");
        amountElm.setText("100.00");
        txnDetailsElm.addContent(amountElm);

        return doc;
}
```

Just as with the .NET example above, this will produce the following XML:

XML generation output

```
<?xml version="1.0" encoding="utf-8"?>
<Request>
  <Transaction>
    <TxnDetails>
      <merchantreference>12345601</merchantreference>
      <amount currency="GBP">100.00</amount>
    </TxnDetails>
  </Transaction>
</Request>
```

Response parsing is also straightforward, making use of the `SAXBuilder` class that builds a document from an `InputStream`. Here we use a `StringReader` to produce an `InputStream` from a `String`, though we will see later that we can bypass this when performing an actual transaction by using the HTTP response stream directly. Note that JDOM requires a suitable underlying XML parser; here we specify the Apache Xerces `SAXParser`.

Example XML parsing

```
private String getDcReference(String response) {

      // Build the document from String
      SAXBuilder builder =
            new SAXBuilder("org.apache.xerces.parsers.SAXParser");
      Document doc = null;
      try {
            StringReader sr = new StringReader(response);
            doc = builder.build(sr);
      }
      catch (Exception e) {
            // handle exception
            return "";
      }

      // Get the root element
      Element root = doc.getRootElement();

      // Get all descendants with the required name
      Iterator iter = root.getDescendants(
            new ElementFilter("datacash_reference"));

      // Assume only one matching element and return its content
      Element dcRef = (Element)iter.next();
      return dcRef.getText();
}
```

Again in a real setting more error catching and handling code would be present. Also, rather than using `getDescendants()` on the root element, which effectively scans the entire document for matching elements, you may prefer to manually traverse the tree using methods such as `getChildren()` iteratively. The JDOM API is well documented though, and should pose no great difficulty in use.

C.2.2. HTTPS Submission

Actual submission is handled by the `HttpURLConnection` class. Note that to use this class for HTTPS (as opposed to HTTP) requires the Java Secure Socket Extensions (JSSE) library to be present; as of Java 1.4.2, this is included with the standard JRE, and code to perform HTTPS requests is no different from HTTP. If you are developing for a version of Java prior to 1.4.2, some additional steps are required. These are beyond the scope of this document, but are easily located through documentation available on the Web.

HTTPS submission

```
// host is a String containing the URL of the Payment Gateway
URL u;
try {
      u = new URL(host);
}
catch (MalformedURLException m) {
      // handle exception
      return;
}
```

```
HttpURLConnection huc;
try {
	// Make the secure socket connection
	huc  = (HttpURLConnection) u.openConnection();
	huc.setRequestMethod("POST");
	huc.setDoOutput(true);
	huc.setDoInput(true);
	huc.connect();

	// Get the request content stream
	OutputStream os = huc.getOutputStream();

	// Write the JDOM document to the request stream
	XMLOutputter xout = new XMLOutputter();
	xout.output(doc, os);
	os.flush();
	os.close();

	// This is the step that actually submits the request
	int code = huc.getResponseCode();

	if (code != 200) {
		// Deal with HTTP error
		return;
	}

	// HTTP success, so build JDOM document from response stream
	SAXBuilder b =
		new SAXBuilder("org.apache.xerces.parsers.SAXParser");
	Document doc;
	try {
		doc = b.build(huc.getInputStream());
	}
	catch (JDOMException e) {
		// Handle exception
		return;
	}
}
catch (IOException i) {
	// handle exception
	return;
}
```

One thing to note here is that we never have to explicitly state the character encoding that we wish to use. Although we didn’t add an XML declaration to the JDOM document when building the request, the `XMLOutputter` object that we use here to write the document to the request stream will add a declaration specifying the encoding that it is using to write to the stream. Therefore performing the XML output directly in this way is safer than first converting it to a `String` before writing to the request stream.

Unfortunately, there are some limitations of the `HttpURLConnection` class. With Java 1.4.2 it is not possible to specify a timeout value or a proxy for a particular instance of the class (later Java versions have added timeout properties to this class, but not proxies). Sun have defined a number of system properties that can be set to achieve these on a per-JVM basis though, so if a proxy or user-defined timeout value are required, these should be set prior to making a connection:

Optional system properties

```
System.setProperty("http.proxyHost", proxyHost);
System.setProperty("http.proxyPort", proxyPort);
System.setProperty("sun.net.client.defaultConnectTimeout", connTime);
System.setProperty("sun.net.client.defaultReadTimeout", readTime);
```

Note that the timeout properties here are specific to the Sun JVM. Other JVMs may specify similar timeout properties. Failing that, as noted above, newer versions of Java have added timeout properties to the `HttpURLConnection` class; you may also consider launching a separate thread to handle the request itself, allowing you to implement timeouts by interrupting the remitter thread after a defined period.

As a final note, in order to support dynamic failover and/or DR procedures, DataCash very much prefer that DNS lookups to the Payment Gateway honour a low “time to live”. Unfortunately, the system property that specifies DNS lookup caching behaviour has been given a default value of -1, or “cache forever” by a number of JVMs (Sun JVMs in particular are known to have this issue). Therefore without the following code, your code will not refresh the IP resolution of the Payment Gateway without restarting the JVM:

Fixing the DNS lookup caching behaviour

```
System.setProperty("networkaddress.cache.ttl", 10);
```

C.3. Perl

Unlike .NET and Java, Perl does not come with any of the required functionality built into a core library. However, several open-source libraries exist and are easily obtainable; indeed some tend to be included in standard distributions of the Perl interpreter. This document was written against Perl 5.8, though we are not aware of any issues using it for other Perl versions.

C.3.1. XML Generation/Parsing

Several libraries for reading and writing XML documents have been implemented in Perl; it is beyond the scope of this document to perform an in-depth comparison of the available options. For the sake of examples though, we will make use of the `XML::LibXML` module (build upon the popular `libxml2` library, which will need to be obtained separately), as it allows both reading and writing of XML documents, and is relatively straightforward to use. Full documentation for the module is readily available, so we will provide only simple examples.

XML generation example

```
#!/usr/bin/perl -w

use strict;
use XML::LibXML;

# Create the document
my $doc = XML::LibXML::Document->new();
$doc->setEncoding("utf-8");

# Add a root element
my $root = $doc->createElement('Request');
$doc->setDocumentElement($root);

# Add some container elements
my $transactionElm = $doc->createElement('Transaction');
$root->appendChild($transactionElm);
my $txnDetailsElm = $doc->createElement('TxnDetails');
$transactionElm->appendChild($txnDetailsElm);

# Add a simple leaf node
my $merchantRefElm = $doc->createElement('merchantreference');
$merchantRefElm->appendTextNode('12345601');
```

```
$txnDetailsElm->appendChild($merchantRefElm);

# Add a leaf node with single attribute
my $amountElm = $doc->createElement('amount');
$amountElm->setAttribute('currency','GBP');
$amountElm->appendTextNode('100.00');
$txnDetailsElm->appendChild($amountElm);
```

Note that when creating the XML document we have explicitly set a character encoding using `setEncoding()`. This has two effects: firstly it will ensure that the appropriate XML declaration is set for the document; secondly it will cause the `toString()` method to use the specified encoding when writing the document to a string. Therefore it doesn't really matter which encoding you choose, since LibXML will ensure that the declaration will match the actual wire-representation. The above code produces the following XML document:

XML generation result

```
<?xml version="1.0" encoding="utf-8"?>
<Request>
  <Transaction>
    <TxnDetails>
      <merchantreference>12345601</merchantreference>
      <amount currency="GBP">100.00</amount>
    </TxnDetails>
  </Transaction>
</Request>
```

Again, reading response documents is easy enough:

XML parsing example

```
sub getDcRef {
  my ($respString) = @_;

  # Build the XML document from string
  my $parser = XML::LibXML->new();
  my $doc = $parser->parse_string( $respString );

  # Get the root element
  my $root = $doc->documentElement();

  # Get all descendants with the required name
  my @nodes = $root->getElementsByTagName( 'datacash_reference' );

  # Assume only one matching node and return its content
  my $dcRef = $nodes[0]->textContent();
  return $dcRef;
}
```

C.3.2. HTTPS Submission

Several libraries for performing HTTPS exchanges are readily available. For the purposes of this document, we will demonstrate use of the LWP library. LWP is a collection of perl modules that collectively provide a simple means of performing most types of web exchanges. Note that in order to perform HTTPS transactions you will additionally require the Crypt::SSLeay library to be installed and available.

The fundamental class upon which LWP is built is the `LWP::UserAgent` class. Instances of this class are essentially “virtual browsers”, so it is this class that we use to submit requests and read responses:

Simple HTTPS request

```
#!/usr/bin/perl w

use strict;
use LWP;

...

sub send_request {
  my ($doc) = @_;

  # Create the UserAgent and a default set of headers
  my $ua = new LWP::UserAgent;
  my $header = new HTTP::Headers;

  # Create the request
  my $endpoint = 'https://testserver.datacash.com/Transaction';
  my $request = new HTTP::Request( POST => $endpoint,
                                   $header, $doc->toString() );

  $ua->timeout(15);

  my $response = $ua->request($request);

  if ($response->is_success()) {
    my $parser = XML::LibXML->new();
    my $responseDoc = $parser->parse_string( $response->content() );
  }
  else {
    # Handle error
  }
}
```

The exact nature of the exchange can be influenced by specifying additional HTTP headers, and through properties of the `LWP::UserAgent` object; this is all well documented. However, a couple of special cases are worth mentioning here.

Firstly, while it is possible to make LWP honour web proxies, the documentation is a little confusing on this. Although `LWP::UserAgent` defines a number of methods for configuring web proxies, these are only suitable for use when performing HTTP requests; LWP does *not* perform proxied HTTPS correctly. Instead, you should first specify proxy settings through a number of environment variables, and then use LWP exactly as though a direct connection is being used:

Defining an HTTPS proxy

```
$proxy = 'http://server';
$ENV{HTTPS_PROXY} = $proxy;
$ENV{HTTPS_PROXY_USERNAME} = user;
$ENV{HTTPS_PROXY_PASSWORD} = pass;

send_request($doc);
```

Secondly, unlike Java and .NET, Perl does not come with a pre-defined set of trusted CA (Certificate Authority) certificates. As such, LWP’s default behaviour when performing HTTPS transactions is to implicitly trust the server certificate it is presented with during SSL handshake. This is a security risk, as it would be possible for a malicious third-party to hijack requests without LWP automatically rejecting their (invalid) server certificate.

It is therefore *highly* recommended when using LWP that this default behaviour is overridden using another environment variable:

Specifying a root certificate path

```
$ENV{HTTPS_CA_FILE} = $path_to_root_cert_pem;
```

With this code in place, you will then need to obtain the root certificates against which all DataCash certificates are signed. Instructions for doing this may be found in the DataCash Support Centre.

C.4. PHP

The code samples below are largely based upon the DataCash PHP API, which was written for PHP 4, though should remain valid for later PHP versions.

C.4.1. XML Generation/Parsing

There are several options for generating/parsing XML documents with PHP. In the most part these tend to be dependent on the version of PHP you are using, so we will not go into any great depth with these, beyond directing you to the PHP Homepage. In particular, for those using up-to-date versions of PHP (5.2 and above), SimpleXML is a straightforward library for reading and writing XML documents that has previously been used successfully to integrate with the DPG. Otherwise the XML Manipulation page documents a number of libraries for generating and parsing XML documents using both PHP 4 and 5; care should of course be taken to check the installation requirements linked from that page to each of the libraries.

C.4.2. HTTPS Submission

Again, several options are available for this functionality. For the sake of this document we will focus on the cURL library, readily available for both PHP 4 and 5. In addition to a version of PHP with cURL support compiled in, you will need to install the `libcurl` library.

Web requests with cURL are based on the creation and use of cURL objects, themselves accessed through cURL handles. The cURL handles are created by calling `curl_init()`, configured via one or more calls to `curl_setup()`, and executed by calling `curl_exec()`:

Simple HTTPS submission example

```
$ch = curl_init();

$host = "https://testserver.datacash.com/Transaction"
curl_setopt ( $ch, CURLOPT_URL, $host );
curl_setopt ( $ch, CURLOPT_POST, 1 );
curl_setopt ( $ch, CURLOPT_POSTFIELDS, $xmlString );
curl_setopt ( $ch, CURLOPT_RETURNTRANSFER, 1 );
curl_setopt ( $ch, CURLOPT_TIMEOUT, $timeoutSeconds );
$response = curl_exec ($ch);
curl_close ($ch);
```

The `curl_setopt()` function admits a wide range of additional options; most of which are beyond the scope of this document. However, we will briefly mention `CURLOPT_PROXY`, and `CURLOPT_PROXYUSERPWD`, which can be used to specify an HTTP proxy server (with optional authentication):

Specifying an HTTP proxy server

```
curl_setopt ( $ch, CURLOPT_PROXY, 'proxy.server.url:port' );
curl_setopt ( $ch, CURLOPT_PROXYUSEPWD, 'user:password' );
```

Finally, while many distributions of cURL come with a standard set of CA root certificates, some do not. If you encounter certificate trust errors when attempting to communicate with the DataCash Payment Gateway (either testserver or live), it is likely that you will need to specify a certificate location:

Specifying a root CA certificate location

```
curl_setopt ( $ch, CURLOPT_CAINFO, $path_to_root_cert_pem );
```

With this code in place, you will then need to obtain the root certificates against which all DataCash certificates are signed. Instructions for doing this may be found in the DataCash Support Centre.