Accessing web services using IBM DB2 for i HTTP UDFs and UDTFs

Nick Lawrence
Yi Yuan

IBM Systems and Technology Group ISV Enablement
March 2013
# Table of contents

**Abstract** .......................................................................................................................... 1  
**Introduction** ...................................................................................................................... 1  
**Sample code** ...................................................................................................................... 1  
**Prerequisites** ..................................................................................................................... 2  
**HTTP overview** .................................................................................................................. 2  
  - Uniform Resource Locator .................................................................................................. 2  
  - HTTP methods .................................................................................................................. 3  
  - POST .............................................................................................................................. 4  
  - GET .............................................................................................................................. 4  
  - PUT .............................................................................................................................. 4  
  - DELETE.......................................................................................................................... 4  
  - HEAD ............................................................................................................................ 4  
  - HTTP methods in a service-oriented architecture .......................................................... 5  
**HTTP request header fields and connection properties** ....................................................... 5  
  - Setting the time out values .............................................................................................. 7  
  - Following redirects ......................................................................................................... 7  
**HTTP response code and header fields** ............................................................................. 7  
**Request message** .............................................................................................................. 8  
**Response message** .......................................................................................................... 9  
**Function reference** .......................................................................................................... 9  
  - SQL HTTP table (verbose) functions ............................................................................. 10  
  - SQL HTTP scalar functions .......................................................................................... 15  
  - SQL helper functions .................................................................................................... 18  
  - Encoding and decoding an HTTP URL .......................................................................... 18  
  - Base64 encoding and decoding ...................................................................................... 19  
**Configuring the JVM** ........................................................................................................ 20  
  - Selecting the JVM .......................................................................................................... 20  
  - JVM options and Java system properties ...................................................................... 20  
  - Using a truststore and keystore for SSL ....................................................................... 21  
  - HTTP proxy support ...................................................................................................... 22  
  - Increasing the JVM heap size ......................................................................................... 23  
**Basic authentication** ......................................................................................................... 23  
**Example scenarios** ............................................................................................................ 24  
  - Load a web resource into the local database ................................................................. 24  
  - Using data obtained from a web service in a join ......................................................... 25  
  - Using a SOAP API ........................................................................................................ 30  
  - Publishing content to a remote server ......................................................................... 32  
  - Processing the response message HTTP header ......................................................... 34  
  - Accessing a web service using basic authentication .................................................... 36  
**Summary** .......................................................................................................................... 39
Abstract
This white paper explains how to access web services using IBM DB2 for i SQL queries and user-defined functions. The paper includes examples that demonstrate how to combine user-defined functions with the built-in SQL/XML support to create the Hypertext Transfer Protocol (HTTP) request and process the HTTP response. The paper discusses the use of these functions to access web services that employ a representational state transfer (REST) design, and web services that employ SOAP in a service-oriented architecture (SOA).

The paper also describes how to communicate with a web service using the Secure Sockets Layer (SSL) and HTTPS protocols.

Introduction
Web services offer exciting opportunities for the software developer and IT architect. Existing web services can be used to rapidly respond to new business requirements as they evolve. In addition, the size and complexity of an IT infrastructure can be reduced by consolidating hardware and software resources into web services that are accessed through standardized interfaces.

In the context of a relational database, a web service can provide resources that are needed by the database to validate or process the data in the database. For example, a web service can return a loan applicant’s credit scores; the database can then use this information to enforce a constraint that requires that only loans for qualified applicants are stored in the database. Another example might be a shipping carrier’s web service that calculates the duration and cost of a prospective shipment; the database could use this service while processing an online sales transaction.

Web services can be used to make a database more active, rather than a traditional passive data store. For instance, a database used by an insurance company could publish suspicious transactions to a web service that scrutinizes the transaction for fraud. This approach is frequently more efficient than a remote application that regularly connects to and queries the local database, because the remote application is notified only when there are meaningful changes in the local database.

Starting with IBM® DB2® for i 7.1 program temporary fix (PTF) Group SF99701 Level 23, user-defined functions and table functions are provided for invoking HTTP methods. As web services are accessed using the HTTP protocol, these new functions make it straightforward for database developers to incorporate web services into an SQL query.

This paper provides a reference for the new functions and provides examples of how they can be used. The paper explores how the built-in XML data type can be effectively combined with these interfaces when communicating with web services. Additionally, the use of basic authentication and SSL for transmitting sensitive information is discussed.

This paper provides only a basic overview of the HTTP protocol, REST architecture, and service-oriented architecture (SOA). For more details, refer to the “Resources” section in this paper.

Sample code
The new HTTP user-defined functions (UDFs) and user-defined table functions (UDTFs) exist in the SYSTOOLS SQL schema.

Accessing web services using IBM DB2 for i HTTP UDFs and UDTFs
SYSTOOLS differs from other DB2 for i supplied schemas in that it is not part of the default system path. When IBM builds general purpose tools or examples, these examples are considered for inclusion in SYSTOOLS. Inclusion in SYSTOOLS gives a wider audience the opportunity to obtain value from these tools.

The tools and examples in SYSTOOLS are considered ready for use, but not part of any IBM product; they are not subject to IBM service and support.

Customers can use the SYSTOOLS routines as it is or as a model to create their own solutions. It is recommended that customers customize a copy of the routines in a different schema. The typical IBM maintenance for SYSTOOLS is to delete existing objects, followed by the creation of the newer versions, without consideration to any modifications made to existing objects.

The Java™ source is provided in the `/QIBM/ProdData/OS/SQLLIB/bin/systools_java_source.jar` file. If the Java source is modified, the new source must be modified so that it exists in a user-defined package (that is not `com.ibm.db2`). This prevents collision between Java packages supplied by IBM and Java packages built by non-IBM developers.

For many environments, the tools in SYSTOOLS might be acceptable to developers as it is, and will not require further enhancements or changes by the customer. In other environments, these tools can be used by a developer to rapidly implement a proof of concept or prototype, before investing in a more robust solution that does exactly what is needed in the most efficient way.

### Prerequisites

In order to use the HTTP UDFs with DB2 for i 7.1, you need to install the following components in your system:

- 5770-SS1 DB2 PTF group SF99701 Level 23
- Java 1.6 or later (5761-JV1 Option 11, 12, 14, or 15)

### HTTP overview

This section defines common terms and concepts used in the documentation of the HTTP UDFs and UDTFs.

**Uniform Resource Locator**

In a RESTful architecture, a Uniform Resource Locator (URL) identifies the resource that the HTTP method will affect. The basic syntax for a URL is:

```
scheme://domain:port/path?query_string
```

The scheme indicates the protocol that is used for sending information. When using the HTTP UDFs and UDTFs in SYSTOOLS, the HTTP and HTTPS schemes are the only two protocols that are relevant.
The domain name (or the IP address) identifies the destination for the URL (for example, www.ibm.com). The domain name portion of a URL is not case sensitive, as DNS ignores character casing when resolving a domain name to an IP address. (In other words, www.EXAMPLE.com is the same as www.example.com)

The port number is optional. If omitted, port 80 is used for the HTTP scheme, and port 443 is used for the HTTPS scheme.

The path is used to find the resource. The path is case sensitive, although some servers might choose to handle the path in a case-insensitive way when locating the requested resource.

If provided, query_string contains one or more name-value pairs (separated by the ampersand symbol) that are provided to software running on the server (for example, first_name=Nick&last_name=Lawrence).

When data needs to be provided as part of a URL (such as in a value of a query string), the data needs to be specially encoded if it contains spaces or special characters. URL encoding can be accomplished using the SYSTOOLS.URLENCODE function, which is discussed later in this paper.

The URL syntax can also include a user ID and password, as shown in the following syntax. Because a URL often appears in system logs and traces, providing a user name and password as part of the URL is generally not a good idea, and a more secure solution is discussed later in this paper.

URL syntax:

```
scheme://userid:pwd@domain:port/path?query_string
```

A link to a formal specification for the syntax and semantics of a URL (RFC 1738) is included in the “Resources” section.

**HTTP methods**

In a RESTful architecture, the URL identifies a resource and the HTTP method indicates what action is to be taken on that resource. This section contains a summary of the most commonly used HTTP methods and their expected behavior. A web service might choose to implement the method differently than what is described here; customers need to refer to the documentation for the service being used to determine the exact behavior of these methods.

In a RESTful architecture, web services can exchange resources (information) using many different representations (XML, JSON, HTML, and so on). For this reason, the HTTP documentation uses the phrase *representation of the resource* when discussing the data that is being transmitted, rather than the term *resource*.
**POST**

The POST method is used to request that the service create a resource as a subordinate of the resource identified by the URL. A representation of the resource to be created is indicated by the request message parameter of the UDF or UDTF.

**GET**

The GET method is used to retrieve a representation of the resource that is identified by the URL. The GET method is expected to be idempotent, meaning that it does not have side effects and will not change the state of the resources on the server.

**PUT**

The PUT method is used to either create or replace a resource. The resource to be modified is identified by the URL. The request message parameter of the UDF or UDTF contains the new representation of the resource.

It is important to understand the difference between the PUT and POST operations. A POST operation always requests the creation of a new resource, multiple POST operations using the same URL and representation result in multiple unique resources being created on the server. The URL for the POST method identifies the resource that contains the new resource.

A PUT operation requests a specific resource (identified by the URL) be replaced, or created if it does not exist. The PUT operation is defined to be idempotent; multiple identical PUT operations will not cause additional state changes.

**DELETE**

The DELETE method deletes the resource (identified by the URL) from the server. This method is also idempotent, meaning that multiple DELETE requests for the same resource do not cause additional state changes on the server.

**HEAD**

The HEAD method behaves similar to the GET method, except that the method returns only the response HTTP header; in other words, a representation of the resource that is identified by the URL is not returned. The response HTTP header contains the response code for the request, and information about the representation that would have been returned if the GET method had been used.
This can be useful when the client needs to know some information about what the response code and header fields for a GET request will be, without actually retrieving the data. For example, a client might issue a HEAD request to determine when a resource was last modified. If a local cached version of a resource is no longer current, then the client application can submit a GET request to retrieve an updated representation of the resource.

The format of the response HTTP header is covered in the “HTTP response code and header fields” section of this paper.

**HTTP methods in a service-oriented architecture**

The HTTP protocol was originally designed to promote RESTful web services. However, the HTTP protocol does not itself require that a web service behave in a RESTful manner, and there are many web services that are designed using SOA.

When a web service is implemented using SOA, the URL identifies an endpoint rather than a resource. The client uses the HTTP POST method, and includes a message that defines a procedure to invoke, in conjunction with the parameters for the procedure call. The web service then returns the output of the procedure to the client in the HTTP response message. Thus, the POST method is used as a Remote Procedure Call (RPC), rather than to create a new resource. The other HTTP methods are usually not relevant in an SOA environment.

The simple object access protocol (SOAP) is an XML-based standard that is often used to define the message that is sent to the web service. An example of constructing a SOAP message and using the HTTP POST method as a Remote Procedure Call is included later in this paper.

**HTTP request header fields and connection properties**

An HTTP request can contain HTTP header fields. The header fields provide the web service some information about the request. For example, a client might indicate that it is sending XML data to the web service. The client might also specify that it prefers to receive XML data in the response.

In addition, the Java virtual machine (JVM) uses a set of connection properties to process the HTTP request. For instance, the connection properties might specify that redirects should be implicitly followed.

The sample HTTP UDFs and UDTFs in SYSTOOLS accept the header fields and connection properties as an HTTPHEADER parameter. The value of this parameter needs to be provided in an XML format. XML is often easier to construct and process (using SQL) than the plain text format that is defined by the HTTP protocol. The HTTP functions in SYSTOOLS convert the XML document into the format required by the HTTP specification.
The value of the HTTPHEADER parameter can be NULL or an empty string, which causes the default properties to be used. An explicit header needs to be supplied only when non-default properties must be used or when header fields must be sent to the server.

Variations of the UDFs have been created so that this value can be supplied as an instance of the XML data type, or as a serialized XML value in a CLOB(10 K).

A sample value for the HTTPHEADER parameter is shown in the following code

```xml
<httpHeader connectTimeout="10"
    followRedirects="true">
    <header name="Accept" value="application/xml" />
    <header name="Content-Type" value="application/xml" />
</httpHeader>
```

Listing 1: Sample HTTPHeader value

In this example, the root httpHeader element includes (optional) attributes that modifies the connection properties. The valid attribute names and values are shown in Table 1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Default</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>connectionTimeout</td>
<td>Integer</td>
<td>System default</td>
<td>Maximum amount of time the JVM will wait for the connection in milliseconds.</td>
</tr>
<tr>
<td>readTimeout</td>
<td>Integer</td>
<td>System default</td>
<td>Maximum amount of time the JVM will wait for reading data in milliseconds.</td>
</tr>
<tr>
<td>followRedirects</td>
<td>true/false</td>
<td>True</td>
<td>Indicates whether redirects should be implicitly followed when a 3xx response code is received from the server.</td>
</tr>
<tr>
<td>useCaches</td>
<td>true/false</td>
<td>True</td>
<td>Instructs the JVM that caches are allowed to be used if available. DB2 for i does not implement a cache in the HTTP UDFs and UDTFs, however, a default cache might be used if the default cache is registered with the JVM.</td>
</tr>
</tbody>
</table>

Table 1: Connection properties

In Listing 1, the header elements supply the name-value pairs that will be sent as header fields to the web service. Each web service supports whatever name-value pairs are relevant for the task it is performing.
You can find a simplified list of some of the most common HTTP request and response header fields using the references in the “Resources” section.

In Listing 1, the `Accept` header instructs the web service to return its response data in an application/xml format. The `Content-Type` header indicates that the application/xml data is sent to the web service.

### Setting the time out values

The `connectionTimeout` and `readTimeout` properties affect the maximum amount of time that the JVM waits to establish a connection or read data. The database and operating system also establish limits on the maximum amount of time that will be spent during socket connections and database function calls. Thus, changing the connection properties does not necessarily cause the UDF or UDTF to wait for the corresponding amount of time before returning an error.

### Following redirects

When the `followRedirects` property is set to true (the default) in the connection properties, the HTTP UDF or UDTF handles a redirection response code (3xx) by automatically resubmitting the request to the URL indicated by the location header field in the response.

There are two scenarios that developers need to be aware of when using this feature.

- A redirect response is **not** implicitly followed when the new location’s URL specifies that a different protocol should be used. In other words, a redirect from `http://www.example.com` to `https://www.example.com` is **not** implicitly followed.

- When the HTTP method is POST, redirects are followed, however, the HTTP method is changed to GET when submitting the request to the new URL.

Web services commonly respond to a successful POST operation with a redirect response code, and a URL to where the true response can be retrieved. Web browsers handle the redirect by performing a GET, using the URL from the redirect. This process avoids problems where the browser reloads the results of a POST request, and inadvertently resubmits the POST. Some more information on this web development design pattern can be found using the references in the “Resources” section.

The HTTP UDFs and UDTFs follow this same convention.

### HTTP response code and header fields

When using the scalar HTTPHEAD function, or one of the verbose table functions, a response HTTP header is provided to the caller of the UDF or UDTF. The response HTTP header includes a response code and header fields. The response code indicates whether the request was successful. The header
fields contain additional information about the response. Similar to the HTTPHEADER input parameter, the response HTTP header is returned in an XML format.

If the UDF or UDTF specifies an HTTPHEADER input parameter that has the CLOB data type, then the response HTTP header will be returned as a serialized XML document in a CLOB. Otherwise, if the HTTPHEADER input parameter has the XML data type, then the response HTTP header will be an instance of the XML data type.

A sample response header is shown in Listing 2.

```xml
<httpHeader responseCode="200">
  <responseMessage>OK</responseMessage>
  <header name="HTTP_RESPONSE_CODE" value="HTTP/1.1 200 OK"/>
  <header name="Server" value="Apache"/>
  <header name="X-Powered-By" value="PHP/5.3.8-2S5.5.0 ZendServer/5.0"/>
  <header name="Transfer-Encoding" value="chunked"/>
  <header name="Date" value="Fri, 04 Jan 2013 23:38:08 GMT"/>
  <header name="Connection" value="close"/>
  <header name="Content-Type" value="application/xml"/>
</httpHeader>
```

Listing 2: Response HTTP header

The response code is included as an attribute of the root httpHeader element; the responseMessage element is a string value that contains the server's explanation for the response code.

Listing 2 is a response HTTP header that has a successful response code (200). The server has returned the text OK as an explanation for the response code.

The HTTP protocol defines ranges of standard values for the response code. The meaning of the HTTP response codes can be determined by using the link in the “Resources” section. When consulting HTTP specifications, the term status code is used instead of response code. In addition, the term reason phrase is used by the specification, whereas the element for the same idea is named responseMessage in Listing 2.

The attributes of the header elements are the name-value pairs for the response's header fields. In Listing 2, the Content-Type header indicates that the returned resource has an application/xml representation. You can find a simplified list of some of the most common HTTP request and response header fields using the link in the “Resources” section.

**Request message**

The HTTP POST and PUT methods require a message to be set to the web service. In a RESTful architecture, this message is a representation of the resource to POST or PUT to the URL.
A non-null value must be supplied for the request message parameter when calling a UDF or UDTF that uses the POST or PUT method.

The HTTP GET and DELETE methods identify the resource to retrieve or delete using only the URL. Thus, the UDFs and UDTFs that invoke an HTTP GET or DELETE will not include a request message parameter.

Note: The request message is referred to as the message body in the HTTP specification. The specification uses the term message to refer to the entire HTTP request (request line, header fields, and message body).

**Response message**

A web service can return a response message for any of the HTTP methods. The structure of the response message is defined by the web service.

If a web service does not return a response message for a particular request, the UDFs and UDTFs return the NULL value as the response message. This commonly happens during a PUT or DELETE request, if the web service does not need to return any information to the client, other than a successful response code in the response HTTP header.

Note: The response message is referred to as the message body in the HTTP specification. The specification uses the term message to refer to the entire HTTP response (status line, header fields, and message body).

**Function reference**

There are many variations of the functions for invoking HTTP methods. Fortunately, all of these functions follow a standard naming convention:

HTTP(method)(data-type)(verbose)

The method component of the name indicates which HTTP method (discussed previously) will be invoked.

The data-type component must be either CLOB or BLOB. This part of the name indicates the data type that will be used for the response message, and (if applicable) the request message.

If a function name ends with the word verbose, it means that it is a table function. The table functions return both the response HTTP headers and the response message as a result set.

For example HTTPPOSTBLOBVERBOSE is table function that uses the POST method to send and receive BLOB data, and HTTPGETCLOB is a scalar function that uses the GET method to retrieve a representation of a resource as a CLOB.
To provide greater flexibility, variations of these functions exist that accept the HTTP method as a parameter, rather than part of the function name (for example, HTTPCLOB or HTTPBLOBVERBOSE). Allowing the HTTP method to be passed as a parameter provides a way to invoke HTTP methods that are rarely used and do not have a UDF or UDTF associated with them. (The OPTIONS or TRACE methods are some examples; these HTTP methods are seldom used and are not discussed in this paper.)

In addition to the naming conventions, the request HTTP header fields and connection properties can be provided to each UDF or UDTF as either an XML data type or as a CLOB that contains serialized XML data. If the UDF or UDTF returns the response HTTP header, then this output value uses the same data type as the data-type request HTTP header.

**SQL HTTP table (verbose) functions**

Table functions that invoke the HTTP methods provide the most verbose response information. These functions return both the response HTTP header and the response message.

There are many scenarios where the result of an HTTP request can only be interpreted by using the information contained in the response HTTP header. For example, if an error happens on the server, the response HTTP header allows the client to determine what kind of problem occurred. The response HTTP header also frequently contains information about the response, such as the content type of the response message.

If a response message is not returned from the web service due to a non-successful response code in the response HTTP header, a NULL value is assigned to the response message column in the result set. In addition, a warning SQLSTATE (01H52) is raised by the UDTF.

If the response HTTP header cannot be retrieved from the remote server due to a connection error, an error SQLSTATE (38000) is raised.

The table functions and their signatures are show in Table 2. Note that all character parameters and columns have a coded character set identifier (CCSID) of 1208 (This corresponds to a UTF-8 encoding). The function names are composed of all uppercase characters; however, Table 2 provides the function name using a mixed case for improved readability.

<table>
<thead>
<tr>
<th>HTTP method</th>
<th>Function name</th>
<th>Input parameter</th>
<th>Input parameter type</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>httpPostBlobVerbose</td>
<td>URL</td>
<td>VARCHAR(2048)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTPHEADER</td>
<td>CLOB(10K)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>REQUESTMSG</td>
<td>BLOB(2G)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Output column</td>
<td>Output column type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RESPONSEMSG</td>
<td>BLOB(2G)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RESPONSEHTTPHEADER</td>
<td>CLOB(10K)</td>
</tr>
<tr>
<td>Function</td>
<td>Input parameter</td>
<td>Input parameter type</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------</td>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td><code>httpPostClobVerbose</code></td>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HTTPHEADER</td>
<td>XML</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REQUESTMSG</td>
<td>BLOB(2G)</td>
<td></td>
</tr>
<tr>
<td>Output column</td>
<td>Output column type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESPONSEMSG</td>
<td>BLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESPONSEHTTPHEADER</td>
<td>XML</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>httpPostClobVerbose</code></td>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HTTPHEADER</td>
<td>CLOB(10K)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REQUESTMSG</td>
<td>CLOB(2G)</td>
<td></td>
</tr>
<tr>
<td>Output Column</td>
<td>Output Column Type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESPONSEMSG</td>
<td>CLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESPONSEHTTPHEADER</td>
<td>CLOB(10K)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>httpGetBlobVerbose</code></td>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HTTPHEADER</td>
<td>XML</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REQUESTMSG</td>
<td>CLOB(2G)</td>
<td></td>
</tr>
<tr>
<td>Output column</td>
<td>Output column type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESPONSEMSG</td>
<td>BLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESPONSEHTTPHEADER</td>
<td>XML</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>httpGetBlobVerbose</code></td>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HTTPHEADER</td>
<td>CLOB(10K)</td>
<td></td>
</tr>
<tr>
<td>Output column</td>
<td>Output column type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESPONSEMSG</td>
<td>BLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESPONSEHTTPHEADER</td>
<td>CLOB(10K)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Accessing web services using IBM DB2 for i HTTP UDFs and UDTFs
### httpGetClobVerbose

<table>
<thead>
<tr>
<th>Function</th>
<th>Input parameter</th>
<th>Input parameter type</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
</tr>
<tr>
<td>HTTPHEADER</td>
<td>CLOB(10K)</td>
<td></td>
</tr>
<tr>
<td>RESPONSEHTTPHEADER</td>
<td>XML</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output column</th>
<th>Output column type</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESPONSEMSG</td>
<td>CLOB(2G)</td>
</tr>
<tr>
<td>RESPONSEHTTPHEADER</td>
<td>CLOB(10K)</td>
</tr>
</tbody>
</table>

### httpGetBlobVerbose

<table>
<thead>
<tr>
<th>Function</th>
<th>Input parameter</th>
<th>Input parameter type</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
</tr>
<tr>
<td>HTTPHEADER</td>
<td>XML</td>
<td></td>
</tr>
<tr>
<td>REQUESTMSG</td>
<td>BLOB(2G)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output column</th>
<th>Output column type</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESPONSEMSG</td>
<td>BLOB(2G)</td>
</tr>
<tr>
<td>RESPONSEHTTPHEADER</td>
<td>CLOB(10K)</td>
</tr>
</tbody>
</table>

### httpPutClobVerbose

<table>
<thead>
<tr>
<th>Function</th>
<th>Input parameter</th>
<th>Input parameter type</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
</tr>
<tr>
<td>HTTPHEADER</td>
<td>XML</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output column</th>
<th>Output column type</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESPONSEMSG</td>
<td>BLOB(2G)</td>
</tr>
<tr>
<td>RESPONSEHTTPHEADER</td>
<td>XML</td>
</tr>
</tbody>
</table>

### httpPutBlobVerbose

<table>
<thead>
<tr>
<th>Function</th>
<th>Input parameter</th>
<th>Input parameter type</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
</tr>
<tr>
<td>HTTPHEADER</td>
<td>XML</td>
<td></td>
</tr>
<tr>
<td>REQUESTMSG</td>
<td>BLOB(2G)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output column</th>
<th>Output column type</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESPONSEMSG</td>
<td>BLOB(2G)</td>
</tr>
<tr>
<td>RESPONSEHTTPHEADER</td>
<td>CLOB(10K)</td>
</tr>
</tbody>
</table>

### Accessing web services using IBM DB2 for i HTTP UDFs and UDTFs
<table>
<thead>
<tr>
<th>Function</th>
<th>Input parameter</th>
<th>Input parameter type</th>
<th>Output column</th>
<th>Output column type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>httpPutClobVerbose</code></td>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td>REQUESTMSG</td>
<td>CLOB(2G)</td>
</tr>
<tr>
<td></td>
<td>HTTPHEADER</td>
<td>XML</td>
<td>RESPONSEMSG</td>
<td>CLOB(2G)</td>
</tr>
<tr>
<td></td>
<td>REQUESTMSG</td>
<td>CLOB(2G)</td>
<td>RESPONSEHTTPHEADER</td>
<td>CLOB(10K)</td>
</tr>
<tr>
<td><code>httpDeleteBlobVerbose</code></td>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td>RESPONSEMSG</td>
<td>BLOB(2G)</td>
</tr>
<tr>
<td></td>
<td>HTTPHEADER</td>
<td>CLOB(10K)</td>
<td>RESPONSEHTTPHEADER</td>
<td>CLOB(10K)</td>
</tr>
<tr>
<td><code>httpDeleteClobVerbose</code></td>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td>RESPONSEMSG</td>
<td>CLOB(2G)</td>
</tr>
<tr>
<td></td>
<td>HTTPHEADER</td>
<td>XML</td>
<td>RESPONSEHTTPHEADER</td>
<td>CLOB(10K)</td>
</tr>
</tbody>
</table>

**Accessing web services using IBM DB2 for i HTTP UDFs and UDTFs**

13
<table>
<thead>
<tr>
<th>Any HTTP Method</th>
<th>httpBlobVerbose</th>
<th>Input parameter</th>
<th>Input parameter type</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTPMETHOD</td>
<td>VARCHAR(128)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTPHEADER</td>
<td>CLOB(10K)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REQUESTMSG</td>
<td>BLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output column</td>
<td>Output column type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESPONSEMSG</td>
<td>BLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESPONSEHTTPHEADER</td>
<td>CLOB(10K)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Any HTTP Method</th>
<th>httpClobVerbose</th>
<th>Input parameter</th>
<th>Input parameter type</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTPMETHOD</td>
<td>VARCHAR(128)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTPHEADER</td>
<td>XML</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REQUESTMSG</td>
<td>CLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output Column</td>
<td>Output Column Type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESPONSEMSG</td>
<td>CLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESPONSEHTTPHEADER</td>
<td>XML</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Any HTTP Method</th>
<th>httpClobVerbose</th>
<th>Input parameter</th>
<th>Input parameter type</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTPMETHOD</td>
<td>VARCHAR(128)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTPHEADER</td>
<td>CLOB(10K)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REQUESTMSG</td>
<td>CLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output column</td>
<td>Output column type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESPONSEMSG</td>
<td>CLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESPONSEHTTPHEADER</td>
<td>CLOB(10K)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Accessing web services using IBM DB2 for i HTTP UDFs and UDTFs
SQL HTTP scalar functions

Although the table functions provide a complete solution for invoking HTTP methods, a scalar version of each HTTP method is provided in order to simplify common queries where the response header is not interesting to the client.

If the response message cannot be returned due to either a connection error or a non-successful response code, an SQL error (SQLSTATE ‘38000’) is raised. The message text may provide some clue as to what caused the error. In the case of a non-successful response code, a better approach for problem analysis is to use the verbose functions and examine the response HTTP header.

Table 3 describes the scalar functions and their signatures. Note that all character parameters and columns have a CCSID of 1208 (This corresponds to a UTF-8 encoding). The function names are composed of all uppercase characters; however, Table 3 provides the function name using mixed-case characters for improved readability.

<table>
<thead>
<tr>
<th>HTTP method</th>
<th>Name</th>
<th>Return type</th>
<th>Parameter</th>
<th>Parameter type</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>httpPostBlob</td>
<td>BLOB(2G)</td>
<td>URL</td>
<td>VARCHAR(2048)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HTTPHEADER</td>
<td>CLOB(10K)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REQUESTMSG</td>
<td>BLOB(2G)</td>
</tr>
<tr>
<td>httpPostBlob</td>
<td></td>
<td>BLOB(2G)</td>
<td>URL</td>
<td>VARCHAR(2048)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HTTPHEADER</td>
<td>XML</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REQUESTMSG</td>
<td>BLOB(2G)</td>
</tr>
<tr>
<td>httpPostClob</td>
<td></td>
<td>CLOB(2G)</td>
<td>URL</td>
<td>VARCHAR(2048)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HTTPHEADER</td>
<td>CLOB(10K)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REQUESTMSG</td>
<td>CLOB(2G)</td>
</tr>
<tr>
<td>Method</td>
<td>Function</td>
<td>Type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td>------</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>httpPostClob</td>
<td>CLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HTTPHEADER</td>
<td>XML</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REQUESTMSG</td>
<td>CLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GET</td>
<td>httpGetBlob</td>
<td>BLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HTTPHEADER</td>
<td>CLOB(10K)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>httpGetBlob</td>
<td>BLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HTTPHEADER</td>
<td>XML</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>httpGetClob</td>
<td>CLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HTTPHEADER</td>
<td>CLOB(10K)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>httpGetClob</td>
<td>CLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HTTPHEADER</td>
<td>XML</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUT</td>
<td>httpPutBlob</td>
<td>BLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HTTPHEADER</td>
<td>CLOB(10K)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REQUESTMSG</td>
<td>BLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>httpPutBlob</td>
<td>BLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HTTPHEADER</td>
<td>XML</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REQUESTMSG</td>
<td>BLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>httpPutClob</td>
<td>CLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HTTPHEADER</td>
<td>CLOB(10K)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REQUESTMSG</td>
<td>CLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>httpPutClob</td>
<td>CLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>URL</td>
<td>VARCHAR(2048)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HTTPHEADER</td>
<td>XML</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REQUESTMSG</td>
<td>CLOB(2G)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>Function</td>
<td>Type</td>
<td>URL</td>
<td>HTTP_METHOD</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>----------</td>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>DELETE</td>
<td>httpDeleteBlob</td>
<td>BLOB(2G)</td>
<td>URL</td>
<td>VARCHAR(2048)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>httpDeleteBlob</td>
<td>BLOB(2G)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>URL</td>
<td>VARCHAR(2048)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HTTPHEADER</td>
<td>CLOB(10K)</td>
</tr>
<tr>
<td></td>
<td>httpDeleteBlob</td>
<td>BLOB(2G)</td>
<td>URL</td>
<td>VARCHAR(2048)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HTTPHEADER</td>
<td>CLOB(10K)</td>
</tr>
<tr>
<td></td>
<td>httpDeleteBlob</td>
<td>BLOB(2G)</td>
<td>URL</td>
<td>VARCHAR(2048)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HTTPHEADER</td>
<td>XML</td>
</tr>
<tr>
<td></td>
<td>httpDeleteClob</td>
<td>CLOB(2G)</td>
<td>URL</td>
<td>VARCHAR(2048)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HTTPHEADER</td>
<td>CLOB(10K)</td>
</tr>
<tr>
<td></td>
<td>httpDeleteClob</td>
<td>CLOB(2G)</td>
<td>URL</td>
<td>VARCHAR(2048)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HTTPHEADER</td>
<td>XML</td>
</tr>
<tr>
<td>HEAD</td>
<td>httpHead</td>
<td>CLOB(10K)</td>
<td>URL</td>
<td>VARCHAR(2048)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HTTPHEADER</td>
<td>CLOB(10K)</td>
</tr>
<tr>
<td></td>
<td>httpHead</td>
<td>XML</td>
<td>URL</td>
<td>VARCHAR(2048)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HTTPHEADER</td>
<td>XML</td>
</tr>
<tr>
<td>Any HTTP method</td>
<td>httpBlob</td>
<td>BLOB(2G)</td>
<td>URL</td>
<td>VARCHAR(2048)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HTTPMETHOD</td>
<td>VARCHAR(128)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HTTPHEADER</td>
<td>CLOB(10K)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REQUESTMSG</td>
<td>BLOB(2G)</td>
</tr>
<tr>
<td></td>
<td>httpBlob</td>
<td>BLOB(2G)</td>
<td>URL</td>
<td>VARCHAR(2048)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HTTPMETHOD</td>
<td>VARCHAR(128)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HTTPHEADER</td>
<td>XML</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>REQUESTMSG</td>
<td>BLOB(2G)</td>
</tr>
<tr>
<td></td>
<td>httpClob</td>
<td>CLOB(2G)</td>
<td>URL</td>
<td>VARCHAR(2048)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HTTPMETHOD</td>
<td>VARCHAR(128)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HTTPHEADER</td>
<td>CLOB(10K)</td>
</tr>
</tbody>
</table>

Accessing web services using IBM DB2 for i HTTP UDFs and UDTFs
SQL helper functions

Scalar functions have been created to help with common encoding issues.

Encoding and decoding an HTTP URL

The URL specification (RFC 1738) defines a set of special characters that need to be replaced with escape sequences (for example, if used in a query string of a URL). The UDFs include functions to perform this encoding and decoding. The signatures of these functions are shown in Table 4. Although these functions support specifying an encoding character set for the URL, the World Wide Web Consortium (W3C) Recommendation states that UTF-8 should be used. Not doing so may introduce incompatibilities.

Table 3: Scalar functions and signatures

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>httpClob</td>
<td>CLOB(2G)</td>
</tr>
<tr>
<td>URL</td>
<td>VARCHAR(2048)</td>
</tr>
<tr>
<td>HTTPMETHOD</td>
<td>VARCHAR(128)</td>
</tr>
<tr>
<td>HTTPHEADER</td>
<td>XML</td>
</tr>
<tr>
<td>REQUESTMSG</td>
<td>CLOB(2G)</td>
</tr>
</tbody>
</table>
### URL encoding/decoding function signature

<table>
<thead>
<tr>
<th>Function name</th>
<th>Result type</th>
<th>Input parameter</th>
<th>Input parameter type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>URLENCODE</td>
<td>VARCHAR(4096)</td>
<td>VALUE</td>
<td>VARCHAR(2048)</td>
<td>Original string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENCODING</td>
<td>VARCHAR(20)</td>
<td>Encoding. If this value is NULL, UTF-8 is used. UTF-8 is recommended by the World Wide Web Consortium (W3C)</td>
</tr>
<tr>
<td>URLDECODE</td>
<td>VARCHAR(4096)</td>
<td>VALUE</td>
<td>VARCHAR(2048)</td>
<td>URL-encoded string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENCODING</td>
<td>VARCHAR(20)</td>
<td>Encoding. If this value is NULL, UTF-8 is used. UTF-8 is recommended by the W3C</td>
</tr>
</tbody>
</table>

**Table 4: URL encoding/decoding function signature**

### Base64 encoding and decoding

Base64 encoding is widely used on the web to represent binary data as a string (for example, when sending hash keys). Functions are provided for encoding and decoding base64 data. Table 5 shows the signatures for the base64 encoding and decoding functions.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Result type</th>
<th>Input parameter</th>
<th>Input parameter type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE64ENCODE</td>
<td>VARCHAR(4096)</td>
<td>IN</td>
<td>VARCHAR(2732) FOR BIT DATA</td>
<td>Original bit string</td>
</tr>
<tr>
<td>BASE64DECODE</td>
<td>VARCHAR(2732) FOR BIT DATA</td>
<td>IN</td>
<td>VARCHAR(4096)</td>
<td>Base64-encoded string</td>
</tr>
</tbody>
</table>

**Table 5: Base64 encode and decode function signatures**
Configuring the JVM

The UDFs and UDTFs are written in Java and run with a JVM. In most scenarios, the UDFs and UDTFs can be used without making any modifications to the JVM configuration. However, there are a few cases where adjustments may need to be made.

- If HTTP proxy support needs to be enabled
- If a truststore or keystore needs to be set up for using SSL
- If the maximum heap size needs to be increased
- If multiple Java Development Kits (JDKs) are installed, and a JVM other than the system default needs to be used

If adjustments are necessary, they need to be made before the first time the JVM is started in the job. In other words, these settings must be in effect the first time the UDF or UDTF is invoked.

Selecting the JVM

When multiple versions of the Java Development Kit (JDK) are installed, the JAVA_HOME environment variable is used to specify which JDK/bit mode to use (and therefore which 5761-JV1 option to use). The variable should be set to the home directory of the JDK. For example, to specify that the 64 bit version of Java 1.6 should be used, the JAVA_HOME environment variable should be set to /QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit.

If the JAVA_HOME environment variable is not set, the default JDK is used. The determination of the default JDK depends on which 5761-JV1 options are installed.

For information on using multiple Java Development Kits on IBM i, please see the link in the references.

JVM options and Java system properties

JVM options and Java system properties determine the environment in which Java programs run.

A Java properties file is one way to set the Java system properties on IBM i. A Java properties looks similar to the one shown in Listing 3. Each line in the file specifies one Java system property and the property’s value.

```
java.library.path=/home/user/lib64
file.encoding=utf-8
```

*Listing 3: Java properties file example*

To specify that a Java properties file should be used by the JVM, the QIBM_JAVA_PROPERTIES_FILE environment variable must be set to the properties file’s path. Listing 4 shows an example of using the CL command to set the environment variable.

```
ADDENVVAR ENVVAR(QIBM_JAVA_PROPERTIES_FILE) VALUE('/home/user/java400/example.properties')
```

*Listing 4: CL command to set properties file environment variable*
If a Java properties file is not specified by the QIBM_JAVA_PROPERTIES_FILE environment variable, a default properties file is used. On IBM i, the JVM looks for a file named, SystemDefault.properties, in the user.home directory of the current user profile. If the properties file is not found in the user.home directory, the JVM uses the properties file in the /QIBM/userdata/java400/ directory.

A properties file can also be used to set the JVM options. Listing 5 shows an example, where Listing 3 has been modified to include a JVM option. The -Xmx2g option is to set the JVM option maximum heap size (-Xmx) to 2 GB.

When you need to set JVM options in the properties file, you need to add #AllowOptions to the first line of the properties file. This syntax indicates that any line beginning with a '-' is treated as a JVM option, rather than as a Java system property.

```
#AllowOptions
-Xmx2g
java.library.path=/home/user/lib64
file.encoding=utf-8
```

**Listing 5: SystemDefault.properties file example**

For more information on setting Java system properties and options, refer to the Setting Java system properties link in the “Resources” section.

For a list of all the JVM options supported on IBM i, refer to the link for JVM command-line options in the “Resources” section. For a more complete list of Java system properties, refer to the link for List of Java system properties in the references.

**Using a truststore and keystore for SSL**

SSL is a protocol for encrypting information over an unprotected network. Hypertext Transfer Protocol Secure (HTTPS), which is a widely used protocol for secure communication on Internet, is layered on top of SSL to add security capabilities to standard HTTP communication.

SSL uses a digital certificate to identify the server or client of the HTTP communication. These certificates are issued from a trusted certificate authority (CA). For more information about obtaining digital certificates and certificate authorities, you can refer to the Digital Certificate Manager link in the “Resources” section.

SSL communication requires a certificate store on the client in order to store certificates for servers that are trusted. The HTTP UDFs and UDTFs take advantage of the Java Secure Socket Extension (JSSE) for SSL communication. JSSE provides the underlying framework for the SSL implementation and uses a truststore and keystore for certificate management.

- The truststore contains the public certificates for remote servers that are trusted by the client. These certificates are obtained from other parties that the client expects to communicate with, or from certificate authorities that the client trusts to identify other parties. This file often has a name such as cacerts, and by default it is located in the Java runtime environment’s (JRE’s) security directory, for example, /QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit/jre/lib/security/cacerts.
The Java Secure Socket Extension (JSSE) package already includes well known certificate authorities in the cacerts file when it is installed. Thus, it is not usually necessary to update the truststore when accessing public websites using the HTTP functions. When the HTTP functions are used in a private network that requires SSL, the certificates for the remote servers may not be included in the default truststore. If an untrusted certificate is received, the HTTP functions fail with a 
java.security.cert.CertPathBuilderException exception error. This problem can be resolved by copying the cacerts file to a different directory and updating the file so that it includes the additional certificates. After adding the server’s certificate to the cacerts file, it is necessary to set the Java system property, javax.net.ssl.trustStore, for the JVM so that the correct cacerts file is used.

- The keystore file is less commonly used and contains only private certificates and keys. The certificates in the keystore are used to verify the client’s identity to the remote server, if the remote server demands client authentication. The client certificate should be added to a keystore file if needed, and the javax.net.ssl.KeyStore system property should be set for the JVM.

The keytool utility can be used to manage the truststore and keystore. You can find more information about this utility using the links in the “Resources” section.

You can get more information about configuring certificate stores by referring to the Secure IBM i with JDBC over SSL link in the “Resources” section. The process of setting up a truststore for a Java application that requires a secure JDBC connection is similar to the process of setting up a truststore for an application that needs to use an SSL connection for the HTTP UDFs and UDTFs.

Table 6 shows a series of Java system properties which can be used to configure the truststore and keystore used for SSL certificates.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property description</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.net.ssl.trustStore</td>
<td>The location of the java truststore file. This contains the collection of CA certificates trusted by JVM (the truststore). The default value is the jssecacerts file or the cacerts file in JRE’s security directory.</td>
</tr>
<tr>
<td>javax.net.ssl.trustStorePassword</td>
<td>The password for the trusted keystore file.</td>
</tr>
<tr>
<td>javax.net.ssl.keyStore</td>
<td>The location of the Java keystore file. This contains an application process’s own certificate and private key.</td>
</tr>
<tr>
<td>javax.net.ssl.keyStorePassword</td>
<td>The password for the keystore file.</td>
</tr>
</tbody>
</table>

Table 6: Java properties for truststore and keystore

**HTTP proxy support**

An HTTP proxy server acts as an intermediary for requests from clients that are seeking resources from Internet or intranet. The HTTP proxy is widely used for security and performance reasons. For example, some HTTP servers are only accessible from certain IP addresses, and clients of the other IP addresses need to look for a proxy server that uses an acceptable address to access the HTTP server.
When using the HTTP UDFs and UDTFs to access Internet resources using proxy servers, a set of Java system properties can be used to set the proxy server. Table 7 lists these properties in detail.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Property description</th>
</tr>
</thead>
<tbody>
<tr>
<td>http.proxyHost</td>
<td>The host name of the proxy server</td>
</tr>
<tr>
<td>http.proxyPort</td>
<td>The port number, the default value being 80.</td>
</tr>
<tr>
<td>http.proxyUser</td>
<td>User name to log on proxy server.</td>
</tr>
<tr>
<td>http.proxyPassword</td>
<td>User password to log on proxy server.</td>
</tr>
<tr>
<td>http.nonProxyHosts</td>
<td>A list of hosts that should be reached directly, bypassing the proxy.</td>
</tr>
</tbody>
</table>

*Table 7: Java properties for proxy server*

Listing 6 shows a properties file example specifying Java system properties for a proxy server.

```
http.proxyHost=www.proxyhost.com
http.proxyPort=8080
http.proxyUser=proxyuser
http.proxyPassword=proxypwd
http.nonProxyHosts=*.ibm.com|wikipedia.org|...
```

*Listing 6: Properties file example for proxy server*

### Increasing the JVM heap size

In some cases, the JVM option maximum heap size (-Xmx) might need to be increased. One example of where this might occur is if a very large file is transferred, and an out-of-memory error occurs. A solution to this problem is to modify the maximum heap size in the properties file. Listing 7 shows an example of modifying the maximum heap size to 2 GB.

```
#AllowOptions
-Xmx2g
```

*Listing 7: SystemDefault.properties file modifying maximum heap size*

If a heap size greater than 3 GB is necessary, a 64-bit JVM must be used. After installing either 5761-JV1 option 12 or 15, the `JAVA_HOME` environment variable can be used to select the appropriate JVM.

### Basic authentication

The HTTP UDFs and UDTFs can supply credentials using basic authentication.

The simplest way to provide the authorization credentials is to supply a user name and password as part of a URL, as shown in the “Uniform Resource Locator” section. Although simple, this approach is not recommended. The problem is that the URL can appear in the SQL message text and job log if an error occurs, thus using this approach can unintentionally expose the password.
A better solution is to encode the credentials into the request header directly. When using basic authorization, a three step process is used to determine the value of the authorization header.

1. The user name and password are combined into a single string separated by a colon `:` (for example, username:password). The string must use the UTF-8 character set.
2. The binary value of the resulting string is encoded in base64.
3. The authorization method (Basic) is put before the encoding string.

An authorization header field for a user that has a name `nick` and a password `password` is shown in Listing 8.

```
<header name="Authorization" value="Basic bmljazpwYXNzdzByZA=="/>
```

Listing 8: Authorization header field

The base64 encoding is not an encryption algorithm, and therefore, handling user names and passwords in SQL requires the following caution:

- Do not specify a password as a string in the source for a program, procedure, or function. Do not specify the password as a string in a view. Instead, use a variable.
- When connected to a remote database, data is not encrypted during the transmission. To protect the password in these cases, consider using a communications encryption mechanism such as Internet Protocol Security Architecture (IPSec) (or SSL if connecting between IBM i products).

An example that uses basic authentication to access a web service is provided later in this paper.

**Example scenarios**

The following example scenarios are included to demonstrate the HTTP functions and describe some potential use cases. Because every web service is different, the documentation of a specific web service should be consulted before using the service. Users of a web service should always read and comply with the web service’s terms of service before using its capabilities.

**Load a web resource into the local database**

In some cases, it might be beneficial to store a resource that is identified by a URL in the local database as a BLOB. This could provide faster or more consistent access to important information that is available on the web.

The example in Listing 9 obtains the PDF file for a document titled *IBM i Strategy and Roadmap* from the IBM website and stores it in a database table as a BLOB. The variable `myURL` is used only to improve readability. The request header used in the HTTPGETBLOB function is the empty string, which causes the default values to be used.
CREATE TABLE PDF(URL VARCHAR(4096),
PDF BLOB(2G));

CREATE VARIABLE myURL VARCHAR(4096);

INSERT INTO PDF(URL, pdf)
VALUES(myURL,
SYSTOOLS.HTTPGETBLOB(myurl, ''));

**Listing 9: Insert a web resource into a database table**

The examples that follow assume that the web service is exchanging XML data with DB2 for i. The XML data allows DB2 for i to do a lot more than retrieval and storage of web resources.

**Using data obtained from a web service in a join**

In the context of a relational database, data that is obtained from a web service is often used in a join with relational data. The web service typically provides additional data that is needed for some processing or analysis.

This example explains a simple analytical scenario. Assuming that a European company performs online business transactions world wide. A large number of sales are performed each business day. In these transactions, customers make payments by converting their currency to Euros as part of the transaction. A summary report for the number of sales (summarized by date and type of currency) is stored in a table named DAILY_SALES. A subset of the data in this table for the United States Dollars (USD) currency is shown in Figure 1.

<table>
<thead>
<tr>
<th>SALES_DATE</th>
<th>SALES_CURRENCY</th>
<th>NUM SALES</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-03-06</td>
<td>USD</td>
<td>184145</td>
</tr>
<tr>
<td>2013-03-05</td>
<td>USD</td>
<td>184202</td>
</tr>
<tr>
<td>2013-03-04</td>
<td>USD</td>
<td>185737</td>
</tr>
<tr>
<td>2013-03-01</td>
<td>USD</td>
<td>186138</td>
</tr>
<tr>
<td>2013-02-28</td>
<td>USD</td>
<td>178930</td>
</tr>
</tbody>
</table>

**Figure 1: DAILY_SALES table**

It has been observed that the number of sales varies from day to day. A possible explanation might be that the number of sales is related to the exchange rate between the Euro and the currency used by the customer for payment. For example, a United States customer might choose to defer making a purchase until the cost of the purchase in USD is less, due to a better exchange rate. In order to verify this theory, historical information on the exchange rate between Euros and USD is needed, but this information is not available in the database.

This example obtains the exchange rates for the last 90 days from a web service sponsored by the European Central Bank. This data is then joined with the existing relational data, shown in Figure 1.

After some initial research, it is possible to determine that the 90-day history of exchange rates can be retrieved from the European Central Bank using the URL shown in Listing 10.

**Listing 10: URL for 90-day exchange rate history**

Listing 11 demonstrates how to retrieve an XML response from the bank using the HTTPGETBLOB function. The first parameter to the function is the URL from Listing 10. The second parameter contains the header fields for the request. In this example, the HTTPHEADER parameter is the empty string, meaning that the default header fields and connection properties will be used. Although many web services would require an Accept header field be specified to indicate that the data must be returned in a particular format, this web service uses an extension (.xml) on the URL indicating that XML data will be returned. Thus, there is no need to specify an explicit Accept header field to request XML content as a response.

Because the HTTPGETBLOB function returns a BLOB that is known to contain a serialized XML document, the BLOB value can be passed into the XMLPARSE function to create an instance of the XML data type.

```sql
VALUES
XMLPARSE (DOCUMENT
SYSTOOLS.HTTPGETBLOB(
  ------------------ URL -----------------------------

  ------------------ Header -----------------------------
  ''
)
)
```

**Listing 11: Retrieve XML response**

A simplified version of the returned XML document is shown in Listing 12. An actual response from this web service contains a cube element for each of the 90 days, with many currency exchange rates for each day. Listing 12 has been shortened so that only two days are shown, with two exchange rates per day.
Relational databases and SQL are designed to work with result sets (rows and columns). The XMLTABLE built-in table function can be used to convert the results from the function call in Listing 11 into an SQL result set. An SQL query that makes use of XMLTABLE is shown in Listing 13.
SELECT my_cube.rate_time, my_cube.currency, my_cube.rate
FROM
XMLTABLE(
--------------- Declare Namespaces ----------------------
XMLNAMESPACES(
DEFAULT 'http://www.ecb.int/vocabulary/2002-08-01/eurofxref',
'http://www.gesmes.org/xml/2002-08-01' AS "gesmes",
),

--------------- Row Expression --------------------------
'gesmes:Envelope/Cube/Cube/Cube'
PASSING
------------ Initial Context ------------------------
XMLPARSE(DOCUMENT
 SYSTOOLS.HTTPGETBLOB{
---------- URL ------------------------
---------- Header ---------------------
''
}
)
--------------- Result Set Columns ----------------
COLUMNS
  currency CHAR(3)        PATH '@currency',
  rate    DECIMAL(10,4)  PATH '@rate',
  rate_time DATE           PATH '../@time'
) my_cube
WHERE currency = 'USD'
ORDER BY rate_time DESC

Listing 13: Using XMLTABLE to create a result set

The XMLTABLE function in Listing 13 has several important components to it.

The XMLNAMESPACES declaration defines two in-scope namespaces. The namespace
http://www.ecb.int/vocabulary/2002-08-01/eurofxref is to be used as the default element
namespace. All unqualified elements in XPath expressions will be qualified by this namespace. The
namespace http://www.gesmes.org/xml/2002-08-01 is bound to the namespace prefix gesmes.

The required result set must include one row for each of the repeating Cube elements that are three Cube
levels deep (The Cube elements with the currency and rate attributes). The row expression selects these
elements to produce the rows of the result set.

In this example, the PASSING clause defines the initial context of the row expression to be the XML
document that is returned from the web service. In other words, the row expression is relative to the root of
the XML document returned from the XMLPARSE function call. The parameter for the XMLPARSE
function is based on Listing 11 and has already been discussed.

The COLUMNS clause defines the columns of the result set. Each column contains an SQL column name,
an SQL type, and an XPath expression that defines how to extract the columns data from the current item
of the row expression. In Listing 13, the currency and rate columns are built from the currency and rate
attributes of the Cube element that has been selected by the row expression. The rate_time column
needs to refer to a time attribute that is in the parent Cube element. Thus, the XPath expression begins
with "." which is an XPath abbreviation for parent::node().
A link to a tutorial for using the XMLTABLE function can be found in the “Resources” section.

The result set from Listing 13 is (partially) shown in Figure 2.

<table>
<thead>
<tr>
<th>RATE_TIME</th>
<th>CURRENCY</th>
<th>RATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-03-06</td>
<td>USD</td>
<td>1.3035</td>
</tr>
<tr>
<td>2013-03-05</td>
<td>USD</td>
<td>1.3034</td>
</tr>
<tr>
<td>2013-03-04</td>
<td>USD</td>
<td>1.3007</td>
</tr>
<tr>
<td>2013-03-01</td>
<td>USD</td>
<td>1.3000</td>
</tr>
<tr>
<td>2013-02-28</td>
<td>USD</td>
<td>1.3129</td>
</tr>
</tbody>
</table>

**Figure 2: Result set from XMLTABLE**

Combining the results in Figure 2 and Figure 1 can now be easily accomplished with INNER JOIN. This is shown in Listing 14, and the INNER JOIN syntax is shown in bold.

```sql
SELECT ds.sales_date, ds.sales_currency, ds.num_sales, my_cube.rate
FROM
XMLTABLE(
  --------------- Declare Namespaces ----------------------
  XMLNAMESPACES(
    DEFAULT 'http://www.ecb.int/vocabulary/2002-08-01/eurofxref',
    'http://www.gesmes.org/xml/2002-08-01' AS "gesmes"
  ),
  ----------------- Row Expression --------------------------
  'gesmes:Envelope/Cube/Cube/Cube'
  PASSING
  --------------- Initial Context --------------------------
  XMLPARSE(DOCUMENT
    SYSTOOLS.HTTPGETBLOB{
      ---------------- Header ---------------------
      ''
    }
  )
  --------------- Result Set Columns -------------------
  COLUMNS
  currency CHAR(3) PATH '@currency',
  rate DECIMAL(10,4) PATH '@rate',
  rate_time DATE PATH '../@time'
) my_cube
INNER JOIN daily_sales ds ON
  (ds.sales_currency = my_cube.currency AND
   ds.sales_date = my_cube.rate_time)
WHERE ds.sales_currency = 'USD'
ORDER BY ds.sales_date DESC
```

**Listing 14: XMLTABLE with INNER JOIN**

The result set for Listing 14 is shown in Figure 3. In this example, the (fictional) number of sales has been fashioned such that it is easier to visually see a strong correlation between the exchange rate and number of sales. In actual applications, more sophisticated statistical techniques can be employed to determine the strength and significance of a correlation.
Using a SOAP API

In response to an event that occurs within the database, the database might need to call upon a web service to carry out an external action. For instance, assume that a banking application needs to offer a service that allows a customer to receive a text message if the customer’s account balance drops below a minimum amount. If the bank has access to a web service that can send a text message, then the database can employ the web service to satisfy the business requirement.

This example assumes that the web service has adopted an SOA design. The remote procedure is invoked by sending a SOAP message to an endpoint (identified by a URL) using the HTTP POST method. It is assumed that the SOAP message must follow the format shown in Listing 15 and that the endpoint URL is http://example.com/WebServices/SMS.asmx.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/
   xmlns:ws="http://www.example.com/WebServices/"
>
   <soap:Body>
      <ws:SendMessage>
         <ws:SMSMessage>
            <ws:MobileNumber>string</ws:MobileNumber>
            <ws:MessageText>string</ws:MessageText>
         </ws:SMSMessage>
      </ws:SendMessage>
   </soap:Body>
</soap:Envelope>
```

**Listing 15: SOAP SendMessages request**

Although this example is fictional, the format of the SOAP message in Listing 15 is based on real web services that provide similar functionality. The example has been simplified to ignore certain issues, such as multiple phone numbers in the same request, billing information, and authorization credentials.

There are two parts to sending the request. The first part is to build the SOAP request that will be sent to the web service as a message. The second part is to send the message to the endpoint’s URL.

Listing 16 defines a function to build the XML value for the SOAP request. The function makes use of the SQL/XML publishing functions. The XMLDOCUMENT and XMLELEMENT publishing functions are used to build the document and element nodes, respectively. The XMLFOREST function is used to create the ws:MobileNumber and ws:MessageText elements as siblings (rather than a nested or parent-child relationship), and to define the contents of those elements. The XMLNAMESPACE declaration is used to define the namespace bindings that are used within the soap:Envelope element.
CREATE FUNCTION build_soap_req(phone VARCHAR(25))
RETURNS XML
LANGUAGE SQL
RETURN
XMLDOCUMENT(
    ---- Soap Envelope ----
    XMLELEMENT(NAME "soap:Envelope",
        XMLNAMESPACES(
            'http://schemas.xmlsoap.org/soap/envelope/' AS "soap",
            'http://www.example.com/WebServices/' AS "ws"
        ),
    ---- Soap Body ----
    XMLELEMENT(NAME "soap:Body",
        XMLELEMENT(NAME "ws:SendMessage",
            XMLELEMENT(NAME "ws:SMSMessage",
                XMLFOREST(phone AS "ws:MobileNumber",
                    'low balance!' AS "ws:MessageText"
                ) -- XMLFOREST
            ) -- ws:SMSMessage
        ) -- ws:SendMessage
    ) -- soap:Body
) -- soap:Envelope
)
Listing 16: Function to build a SOAP request

The document that is produced by the function matches the document in Listing 15 with the exception that the content of the ws:MobileNumber and ws:MessageText elements are set to concrete values.

Listing 17 shows an SQL statement where the scalar HTTPPOST BLOB function is used to post the XML data constructed by the function defined in Listing 16. The SOAPAction and Content-Type header fields are added to the request's header. The SOAPAction header field is part of the SOAP standard; it is used by the web service to filter HTTP requests without actually parsing the message. For this example, the test team made the assumption that the SOAPAction header field needs to have a value of http://www.example.com/WebServices/SendMessages.

The response from the web service is stored in the post_blob_response variable.

Accessing web services using IBM DB2 for i HTTP UDFs and UDTFs

31
CREATE VARIABLE post_blob_response BLOB;

CREATE VARIABLE notify_phone_number VARCHAR(25) DEFAULT '1234567890';

SET post_blob_response = SYSTOOLS.HTTPPOSTBLOB(
  -- URL --
  ' http://example.com/WebServices/SMS.asmx',

  -- Header --
  '<httpHeader>
    <header name="SOAPAction" value="http://www.example.com/WebServices/SendMessages"/>
    <header name="Content-Type" value="application/soap+xml"/>
  </httpHeader>',

  -- Message --
  XMLSERIALIZE(build_soap_req(notify_phone_number) AS BLOB(2G))
)

Listing 17: HTTPPOSTBLOB invocation

The statements in Listing 17 can easily be embedded in a trigger or stored procedure to accomplish the business requirement.

This example has shown that the HTTP methods can be used to access web services that are designed to use a service-oriented architecture, instead of a resource-oriented architecture. It also demonstrates how a web service can be used to make the database more active by performing external actions when database events occur.

Publishing content to a remote server

In some cases, the database may need to update a web resource with new or changed data. Assume that an insurance company operates many data centers that manage the day-to-day operations of the company. Although each data center has a unique application environment, the company has consolidated their fraud detection applications into a central web service. At the end of each business day, each data center must submit claim information (in an XML format) for claims that have been created or updated that day and are for amounts greater than $50,000. The data is to be submitted to a URL that identifies the branch and date of the submission. For instance, a URL that can be used to submit claims for branch_xyz on February 8, 2013 might look as the one shown in Listing 18.

http://example.com/branch_xyz/updated_claims/20130208

Listing 18: Example URL

The information is to be submitted using the HTTP PUT method, so that a list of claims identified by the URL is updated if the list already exists.
Figure 4 shows a CLAIMS table containing the claims that have been created or updated in branch_xyz local data center on February 8, 2013. Several claims are for amounts more than $50,000 and need to be sent to the central web service, using the URL mentioned in Listing 18.

<table>
<thead>
<tr>
<th>ID</th>
<th>CLAIMANT</th>
<th>AMOUNT</th>
<th>LAST UPDATED</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234567890</td>
<td>Nick</td>
<td>75000</td>
<td>2013-02-08 13:06:35.653674</td>
</tr>
<tr>
<td>1234567891</td>
<td>Bob</td>
<td>49000</td>
<td>2013-02-08 13:06:35.653674</td>
</tr>
<tr>
<td>1234567892</td>
<td>George</td>
<td>88000</td>
<td>2013-02-08 13:06:35.653674</td>
</tr>
<tr>
<td>1234567893</td>
<td>Grace</td>
<td>25000</td>
<td>2013-02-08 13:06:35.653674</td>
</tr>
<tr>
<td>1234567894</td>
<td>Kim</td>
<td>99000</td>
<td>2013-02-08 13:06:35.653674</td>
</tr>
</tbody>
</table>

For simplicity, it has been assumed that the XML document that must be sent to the web service should look as shown in Listing 19.

```
<?xml version="1.0" encoding="UTF-8"?>
<daily_update>
  <claim>
    <id>1234567890</id>
    <claimant>Nick</claimant>
    <amount>75000</amount>
    <update_time>2013-02-08T13:06:35.653674</update_time>
  </claim>
  <claim>
    <id>1234567892</id>
    <claimant>George</claimant>
    <amount>88000</amount>
    <update_time>2013-02-08T13:06:35.653674</update_time>
  </claim>
  <claim>
    <id>1234567894</id>
    <claimant>Kim</claimant>
    <amount>99000</amount>
    <update_time>2013-02-08T13:06:35.653674</update_time>
  </claim>
</daily_update>
```

Listing 19: Daily update XML document

Building the XML document as in Listing 19 might sound complicated, but the document can be built with a straightforward application of the XMLGROUP function.

Listing 20 shows how an SQL query using the aggregate XMLGROUP publishing function, can be used to transform relational rows and columns into the XML document shown in Listing 19. The AS clause is used to assign an element name for the columns that have been provided as parameters. Each row in the aggregation becomes a claim element; the claim elements then become the children of the root daily_update element. As there is no GROUP BY clause used with the select, there is only one row (and therefore one XML document) in the result set.
SELECT XMLGROUP(id AS "id",
                claimant AS "claimant",
                amount AS "amount",
                last_updated AS "update_time"
        OPTION ROW "claim"
        ROOT "daily_update"
    ) AS summary_doc
FROM claims
WHERE amount > 50000 AND
    DATE(last_updated) = '2013-02-08'

Listing 20: Query to build the daily update XML document

The query in Listing 20 results in a scalar value, and this means it can be used as a parameter of the HTTPPUTBLOB function. Listing 21 shows how this is accomplished. The query has been modified to serialize the XML value to a BLOB, as a BLOB data type is required by the HTTPPUTBLOB function. The response from the web service is stored in a variable named put_response for use by the application.

CREATE VARIABLE put_response BLOB(2G);

SET put_response =
    SYSTOOLS.HTTPPUTBLOB(
        --- URL ---
        'http://example.com/branch_xyz/updated_claims/20130208',
        --- Header ---
        '<httpHeader>
        <header name="Content-Type" value="application/xml"/>
        </httpHeader>',
        --- Message ---
        (SELECT XMLSERIALIZE(
            XMLGROUP(id AS "id",
                        claimant AS "claimant",
                        amount AS "amount",
                        last_updated AS "update_time"
                OPTION ROW "claim"
                ROOT "daily_update"
            ) AS BLOB(2G)) AS summary_doc
        FROM claims
        WHERE amount > 50000 AND
            DATE(last_updated) = '2013-02-08'
    )

Listing 21: HTTP PUT BLOB function

Processing the response message HTTP header

Assume that it is necessary to verify that the HTTPPUTBLOB in Listing 21 was successful. Further, assume that the web service accepts the data provided and does not return any content in the response message. In that case, the put_response variable will be assigned the NULL value. The response HTTP header that is returned from the HTTPPUTBLOBVERBOSE function must be used to determine the status of the request.
Listing 22 shows how a CROSS JOIN can be used to pass the RESPONSEHTTPHEADER column from the HTTPPUTBLOBVERBOSE table function as a parameter of the XMLTABLE table function. The XMLTABLE function extracts the response code and response message text from the response HTTP header. The final SELECT includes all the columns returned from XMLTABLE, and the response message returned from the HTTPPUTBLOBVERBOSE table function.

```sql
SELECT xt.*,
       put_blob_rs.responseMsg
FROM
  TABLE(
    SYSTOOLS.HTTPPUTBLOBVERBOSE(
      --- URL ---
      'http://example.com/branch_xyz/updated_claims/20130208',
      --- Header ---
      '<httpHeader>
        <header name="Content-Type"
          value="application/xml"/>
      </httpHeader>',
      --- Message ---
      (SELECT
         XMLSERIALIZE(
           XMLGROUP(id           AS "id",
                      claimant     AS "claimant",
                      amount       AS "amount",
                      last_updated AS "update_time"
          OPTION ROW "claim"
                      ROOT "daily_update"
        ) AS BLOB(2G)) AS summary_doc
      FROM claims
      WHERE amount > 50000 AND
      DATE(last_updated) = '2013-02-08'
      ) put_blob_rs
  CROSS JOIN
  XMLTABLE('httpHeader'
    PASSING
    XMLPARSE(DOCUMENT put_blob_rs.responseHttpHeader)
    COLUMNS
    code INTEGER PATH '@responseCode',
    message VARCHAR(200) PATH 'responseMessage'
  ) xt;
```

Listing 22: Response HTTP header cross join with XMLTABLE

The result of the CROSS JOIN is shown Figure 5. An HTTP response code of 204 indicates that the request was successful, but the server did not need to return any content. The NULL value ('-') is returned for the response message.
Accessing a web service using basic authentication

Business processes often need to access resources that require credentials using basic authentication. The popular Gmail web service is used in this example to illustrate how to accomplish this. This service was chosen because it is well known, it requires basic authentication, and it returns XML data.

The first step is to assign the user name and password to a global variable. Using a global variable instead of a literal string helps prevent these sensitive pieces of information from appearing in the system catalogs if the password is ever used as part of a function, procedure, or view. A UTF-8 character set is used for the variable; when this value is converted to base64, the base64 encoding needs to be based on a binary version of UTF-8 data.

```
CREATE VARIABLE mypassword VARCHAR(1024) CCSID 1208;
SET mypassword = 'username:password';
```

Listing 23: Global variable to store user ID and password

Next, the request header is constructed. If there are many header fields, or if the header fields depend on relational data, it might be easier to build the header XML document using the SQL/XML publishing functions. Listing 24 shows how an SQL query is used to assign the request header to a global variable called `header_data`. A list of name and value pairs is supplied as the table for the query. The authorization header field’s value is calculated using the SYSTOOLS.BASE64ENCODE function to encode the user name and password into the base64 format. The XMLGROUP function is used in the SELECT statement to convert the result set into an XML document. The AS ATTRIBUTES clause causes the columns to be created as attributes (rather than elements) in the XML document.
CREATE VARIABLE header_data XML;

SET header_data = (  
SELECT     
XMLGROUP(requestHeader.hname AS "name",  
          requestHeader.hvalue AS "value"  
OPTION ROW "header"  
    ROOT "httpHeader"  
    AS ATTRIBUTES) AS header  
FROM     
(VVALUES  
    -- Authorization header ---  
    ('Authorization', 'Basic ' || SYSTOOLS.BASE64ENCODE(mypassword) ),  
    -- Accept header --  
    ('Accept', 'application/atom+xml' )  
) requestHeader(hname, hvalue)
);

Listing 24: Constructing the request header

The value that is assigned to the header_data variable is shown in Listing 25.

<httpHeader>  
<header name="Authorization" value="Basic bmljazpwYXNzdXJhY2VzZQ=="/>  
<header name="Accept" value="application/atom+xml"/>  
</httpHeader>

Listing 25: header_data

The header information can now be provided to the HTTPGETBLOB function. Because XML data is returned from the web service, the XMLTABLE function is used to decompose the data into rows and columns.
SELECT result.*
FROM
XMLTABLE(
    XMLNAMESPACE(DEFAULT 'http://purl.org/atom/ns#'),
    'feed/entry'
PASSING
    XMLPARSE(DOCUMENT
        SYSTOOLS.HTTPGETBLOB{
            -- URL --
            'https://mail.google.com/mail/feed/atom/',
            -- header --
            header_data
        }
    )
)
COLUMNS
    issued     TIMESTAMP PATH 'issued',
    title      VARCHAR(128) PATH 'title',
    author_name VARCHAR(255) PATH 'author/name'
) AS result

Listing 26: Query to access atom feed with basic authentication

The results of the query are shown in Figure 6.

<table>
<thead>
<tr>
<th>ISSUED</th>
<th>TITLE</th>
<th>AUTHOR_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-02-10 04:...</td>
<td>Google Account password changed</td>
<td>accounts-nospam</td>
</tr>
<tr>
<td>2013-02-10 03:...</td>
<td>Google Email Verification</td>
<td>account-verification</td>
</tr>
<tr>
<td>2013-02-07 00:...</td>
<td>Getting started on Google+</td>
<td>Google+ team</td>
</tr>
<tr>
<td>2013-02-07 00:...</td>
<td>Customize Gmail with colors a...</td>
<td>Gmail Team</td>
</tr>
<tr>
<td>2013-02-07 00:...</td>
<td>Get Gmail on your mobile phone</td>
<td>Gmail Team</td>
</tr>
<tr>
<td>2013-02-07 00:...</td>
<td>Import your contacts and old</td>
<td>Gmail Team</td>
</tr>
</tbody>
</table>

Figure 6: Result set
Summary

This paper has provided reference for how to use the HTTP functions, and how to combine them with the XML support that is available in DB2 for i 7.1.

When the new tools in the SYSTOOLS schema are combined with the built-in XML support available in DB2 for i, developers and architects have a tremendous opportunity to use the web services within SQL. These new functions are simple to use, and yet adaptable enough to tackle many challenges. Customers who prefer to use routines written in external languages such as RPG or C (as opposed to Java) can still find these functions useful for initial development and exploration of web services, before investing in a solution that meets the exact needs of the business.
Resources

The following websites provide useful references to supplement the information contained in this paper:

- Accessing RESTful services from DB2: Introducing the REST user-defined functions for DB2. (This article talks about the support for these functions on DB2 for z/OS and DB2 for LUW)
  ibm.com/developerworks/data/library/techarticle/dm-1105htprestdb2/

- RESTful Web services: the basics
  ibm.com/developerworks/webservices/library/ws-restful/

- SOA and web services on developerWorks
  ibm.com/developerworks/webservices/

- XMLTABLE tutorial
  http://pic.dhe.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzasp%2Frbafyxmlte xample.htm

- Tutorial for SQL/XML publishing functions
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzasp%2Frba fyxml3909.htm

- Replacing XML Extender with the integrated SQL/XML support
  ibm.com/partnerworld/page/stg_ast_sys_wp_db2_xml_extender_capabilities

- Using RPG to exploit DB2 XML Support
  ibm.com/developerworks/ibmi/library/i-using-rpg/index.html

- Getting started with the XML Data Type Using DB2 for IBM i
  http://www.ibmsystemsmag.com/ibmi/developer/general/xml_db2_part1/
- Using XML with DB2 for IBM i
  http://www.ibmsystemsmag.com/ibmi/developer/general/xml_db2_part2/

- Now Introducing XML in SQL on DB2 for IBM i!
  http://www.mcpressonline.com/sql/now-introducing-xml-in-sql-on-db2-for-ibm-i.html

- DB2 for IBM i Technology updates
  ibm.com/developerworks/mydeveloperworks/wikis/home?lang=en#/wiki/IBM%20i%20Technology%20Updates/page/DB2%20for%20i%20-%20Technology%20Updates

- Hypertext Transfer Protocol (RFC 2616)
  http://www.w3.org/Protocols/rfc2616/rfc2616.html

- HTTP Status Codes (RFC 2616 section 10)
  http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

- URL Specification (RFC 1738)
  http://www.ietf.org/rfc/rfc1738.txt

- W3C Recommendation for non-ASCII characters in URL Attribute Values
  http://www.w3.org/TR/html40/appendix/notes.html#non-ascii-chars

- Common HTTP header fields

- Information on the Post/Redirect/Get web development design pattern
  http://en.wikipedia.org/wiki/Post/Redirect/Get

- Atom Standard
  http://en.wikipedia.org/wiki/Atom_%28standard%29
- JVM command-line options

- Using multiple Java Developer Kits on IBM i
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzaha%2Fmultjdk.htm

- Setting Java system properties
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzaha%2Fsysprop.htm

- List of Java system properties
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzaha%2Fsysprop2.htm

- Digital Certificate Manager
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzahu%2Frzahudigitalcertmngmnt.htm

- IBM i keytool utility
  http://pic.dhe.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzahz%2Frzahzkeytool.htm

- IBM Java V6 security information for the keytool utility

- Secure IBM i with JDBC over SSL
  http://www.ibmsystemsmag.com/ibmi/administrator/security/secure_ssl/?page=1
About the authors

Nick Lawrence is an Advisory Software Engineer working on DB2 for i in IBM Rochester. He has been involved with DB2 for i since 1999. His most recent responsibilities have been in the area of full text search, SQL/XML, and XMLTABLE. You can reach Nick at ntl@us.ibm.com.

Yi Yuan is a software developer in DB2 team in CSTL (China System and Technology Lab). Yi has been working on XML new features of DB2 for i since 2009. Before that, Yi worked on development of IBM OmniFind® Text Search Server for DB2 for i. You can reach Yi at cdlyuany@cn.ibm.com.
Accessing web services using IBM DB2 for i HTTP UDFs and UDTFs
Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.