UTL HTTP

In the UTL_HTTP section of Appendix A on *Necessary Supplied Packages*, I mentioned that the package body source code for an improved version of the UTL_HTTP package, HTTP_PKG, was not included in the chapter, due to size constraints. Here, I continue from the chapter including the relevant code and added comments. The new UTL_HTTP package will support the 'old fashioned' UTL_HTTP interface of REQUEST and REQUEST_PIECES, but will also add support for:

- Getting the HTTP headers back with each request. These headers contain useful information such as the status of the request (for example, 200 OK, 404 Not Found, and so on), the name of the server that executed the URL, the content type of the returned content, cookies, and so on.
- Getting the content back as either a CLOB or a BLOB. This allows your PL/SQL to retrieve a large PDF to be inserted into a table indexed by interMedia, as well as retrieve plain text from a web page, or just to access any binary data returned by a web server.
- Perform a HEAD of a document. This is useful to check and see if the document you retrieved last week has been updated for example.
- URL-encoded strings. For example, if you have a space or a tilde (~) in a URL request, they must be 'escaped'. This function escapes all characters that need to be.
- Sending cookies with requests. If you are using this HTTP_PKG to get access to a web site that uses a cookie-based authentication scheme, this is crucial. You would have to GET the login page, sending the username and password in the GET request. You would then look at the HTTP headers returned by that page and extract their cookie. This cookie value is what you need to send on all subsequent requests to prove who you are.
- POST data instead of just GET. The GET protocol has limits that vary by web server on the size of the request. Typically, a URL should not exceed 1 to 2 KB in length. If it does, you should POST the data. POSTed data can be of unlimited size.

The amazing thing is we can implement all of this in PL/SQL using our SocketType from the

UTL_TCP section, and we can do it in a fairly small amount of code.

To begin, we will add some more object types to our library. These object types are specifically designed to hold a correlated array – that is name/value pairs. We see these name/value pairs in many Internet protocols. In SMTP (simple mail transfer protocol) we see them in the form of From: someone@somewhere.com and Subject: This is my email. In NNTP (Net News Transfer Protocol) they are used heavily in the message header. In HTTP, the protocol we are getting ready to implement, they are again used. In HTTP, every request and response begins with a series of lines that are name/value pairs. In order to make dealing with these name/value pairs easy, I've developed a small object type to handle them. The implementation is as follows:

```
ops$tkyte@DEV816> create or replace type CorrelationType as object
  2  (    name    varchar2(50),
  3       value    varchar2(8192)
  4  )
  5  /

Type created.

ops$tkyte@DEV816> create or replace type CorrelationArrayType
  2  as varray(255) of CorrelationType
  3  /

Type created.

ops$tkyte@DEV816> create or replace type CorrelatedArray as object
  2  (
  3       -- These are an array of our name/value pairs.
  4       -- Our methods below operate on this array.
  5       vals    CorrelationArrayType,
  6
  7       -- Given a name, this searches the above varray
  8       -- for its value, returning Null if no value is found.
  9       member function valueof( p_name in varchar2 ) return varchar2,
 10
 11       -- Adds a name/value pair to the end of the array.
 12       -- Does NOT check to see if that name already exists!
 13       -- Use with caution, because if that name already exists, this
 14       -- will not overwrite it (and if you call valueof, you'll
 15       -- get the OLD value at the front of the array).
 16       member procedure addpair(p_name in varchar2, p_value in varchar2),
 17
 18       -- Add header line parses a string in the format:
 19       -- NAME: value
 20       -- which is the typical internet style header.
 21       -- This uses addpair above.
 22       member procedure addHeaderline( p_line in varchar2 ),
 23
 24       -- Updates an existing entry or if none exists, calls
 25       -- addpair to add it to the end.
 26       member procedure updateValue( p_name in varchar2,
 27                                     p_value in varchar2 )
 28  )
 29  /

Type created.
```

That's the specification for our object type `CorrelatedArray`. As you can see, we needed to create two 'helper' object types; the correlation element, a single name/value pair, and an array of these name/value pairs. Our final object type has this array embedded in it and a series of methods to operate on that array type.

And now for the body:

```
tkyte@TKYTE816> create or replace type body CorrelatedArray as
  2
  3    member procedure
  4    addpair( p_name in varchar2, p_value in varchar2 )
  5    is
  6    begin
  7      vals.extend;
  8      vals( vals.count ) := CorrelationType( upper(p_name), p_value );
  9    end addpair;
 10
 11    member procedure
 12    updateValue( p_name in varchar2, p_value in varchar2 )
 13    is
 14      l_found boolean := FALSE;
 15    begin
 16      for i in 1 .. vals.count loop
 17        if vals(i).name = p_name then
 18          vals(i).value := p_value;
 19          l_found := TRUE;
 20          exit;
 21        end if;
 22      end loop;
 23      if not l_found then
 24        addpair( p_name, p_value );
 25      end if;
 26    end updateValue;
 27
 28    member function
 29    valueof( p_name in varchar2 )  return varchar2
 30    is
 31      l_name    varchar2(50) default upper(p_name);
 32      l_idx    number default 1;
 33    begin
 34      for i in 1 .. vals.count loop
 35        exit when ( vals(i).name = l_name );
 36        l_idx := l_idx+1;
 37      end loop;
 38
 39      if ( l_idx <= vals.count ) then
 40        return vals(l_idx).value;
 41      else
 42        return NULL;
 43      end if;
 44    end valueof;
 45
 46    member procedure
 47    addHeaderLine( p_line in varchar2 )
 48    is
 49      l_n number default instr(p_line,':');
```

```
 50    begin
 51      addpair( trim(substr( p_line, 1, l_n-1 )),
 52               trim(substr(p_line,l_n+1)) );
 53    end addHeaderLine;
 54
 55  end;
 56  /

Type body created.
```

The specification for our new HTTP_PKG package follows. We'll look at the specification and some examples that use it. The first two functions, REQUEST and REQUEST_PIECES, are functionally equivalent (minus SSL support) to functions found in the UTL_HTTP package from version 7.3.x, 8.0.x, and 8.1.5. We even will raise the same sort of named exceptions that they would:

```
tkyte@TKYTE816> create or replace package http_pkg
  2  as
  3      function request( url in varchar2,
  4                        proxy in varchar2 default NULL )
  5      return varchar2;
  6
  7      type html_pieces is table of varchar2(2000)
  8         index by binary_integer;
  9
 10      function request_pieces(url in varchar2,
 11                              max_pieces natural default 32767,
 12                              proxy in varchar2 default NULL)
 13      return html_pieces;
 14
 15      init_failed exception;
 16      request_failed exception;
```

The next procedure is GET_URL. It invokes the standard HTTP GET command on a web server. The inputs are:

- p_url – The URL to retrieve.
- p_proxy – The name:<port> of the proxy server to use. NULL indicates that you need not use a proxy server. Some examples are: p_proxy => 'www-proxy' or p_proxy => 'www-proxy:80'.
- p_status – This is returned to you and will be the HTTP status code returned by the web server. 200 indicates normal, successful completion. 401 indicates unauthorized, and so on.
- p_status_txt – This is returned to you and contains the full text of the HTTP status record. For example it might contain: HTTP/1.0 200 OK.
- p_httpHeaders – These may be set by you and upon return will contain the HTTP headers from the requested URL. On input, any values you have set will be transmitted to the web server as part of the request. On output, the headers generated by the web server will be returned to you. You can use this to set and send cookies or basic authentication or any other HTTP header record you wish.
- p_content – A *temporary* CLOB or BLOB (depending on which overloaded procedure

you call) that will be allocated for you in this package (you need not allocate it). It is a session temporary LOB. You may use dbms_lob.freetemporary to de-allocate it whenever you want, or just let it disappear when you log out.

```
18      procedure get_url( p_url        in varchar2,
19                         p_proxy      in varchar2 default NULL,
20                         p_status           out number,
21                         p_status_txt       out varchar2,
22                         p_httpHeaders   in out CorrelatedArray,
23                         p_content    in out clob );
24
25      procedure get_url( p_url        in     varchar2,
26                         p_proxy      in     varchar2 default NULL,
27                         p_status           out number,
28                         p_status_txt       out varchar2,
29                         p_httpHeaders   in out CorrelatedArray,
30                         p_content    in out blob );
```

The next procedure is HEAD_URL, which invokes the standard HTTP HEAD syntax on a web server. The inputs and outputs are identical to GET_URL above (except *no* content is retrieved). This function is useful to see if a document exists, what its mime-type is, or if it has been recently changed, without actually retrieving the document itself:

```
32      procedure head_url( p_url        in varchar2,
33                          p_proxy      in varchar2 default NULL,
34                          p_status        out number,
35                          p_status_txt       out varchar2,
36                          p_httpHeaders   out CorrelatedArray );
```

The next function, URLENCODE, is used when building GET parameter lists or building POST CLOBs. It is used to escape special characters in URLs (for example, a URL may not contain a blank, a URL may not contain a % sign, and so on). Given input such as, Hello World, URLENCODE will return Hello%20World:

```
38      function urlencode( p_str in varchar2 ) return varchar2;
```

The procedure ADD_A_COOKIE allows you to easily set a cookie value to be sent to a web server. You need only know the name and value of the cookie. The formatting of the HTTP header record is performed by this routine. The p_httpHeaders variable you send in/out of this routine would be sent in/out of the <Get|Head|Post>_url routines:

```
40      procedure add_a_cookie( p_name in varchar2,
41                              p_value in varchar2,
42                              p_httpHeaders in out CorrelatedArray );
```

The next procedure, SET_BASIC_AUTH, allows you to enter a username/password to access a protected page. You need only know the name and password to be used. The formatting of the HTTP header record is performed by this routine. The p_httpHeaders variable you send in/out of this routine would be sent in/out of the <Get|Head|Post>_url routines as well:

```
44      procedure set_basic_auth( p_username in varchar2,
45                                p_password in varchar2,
```

```
   46                                  p_httpHeaders in out CorrelatedArray );
```

The procedure SET_POST_PARAMETER is used when retrieving a URL that needs a large (> 2000 or so bytes) set of inputs. It is recommended that the POST method be used for large requests. This routine allows you to add parameter after parameter to a POST request. This POST request is built into a CLOB you supply:

```
   48    procedure set_post_parameter( p_name in varchar2,
   49                                  p_value in varchar2,
   50                                  p_post_data in out clob,
   51                                  p_urlencode in boolean default FALSE );
```

The next two routines are identical to GET_URL above with the addition of the p_post_data input. p_post_data is a CLOB built by repeated calls to set_post_parameter above. The remaining inputs/outputs are defined the same as they were for GET_URL:

```
   53      procedure post_url( p_url        in     varchar2,
   54                          p_post_data  in     clob,
   55                          p_proxy      in     varchar2 default NULL,
   56                          p_status           out number,
   57                          p_status_txt       out varchar2,
   58                          p_httpHeaders   in out CorrelatedArray,
   59                          p_content    in out clob );
   60
   61      procedure post_url(p_url        in     varchar2,
   62                         p_post_data  in     clob,
   63                         p_proxy      in     varchar2 default NULL,
   64                         p_status           out number,
   65                         p_status_txt       out varchar2,
   66                         p_httpHeaders   in out CorrelatedArray,
   67                         p_content    in out blob );
   68
   69  end;
   70  /

  Package created.
```

So, the specification of the package is done – the procedures defined within are rather straightforward. I can do things like GET_URL, which will get a URL. This will use the HTTP GET syntax to retrieve the contents of a web page into a temporary BLOB or CLOB. I can HEAD_URL, which gets the headers for a URL. Using that, I could look at the mime-type for example, to decide if I wanted to use a CLOB (text/html) to get the URL or a BLOB (image/gif). I can even POST_URL to post large amounts of data to a URL. There are the other helper functions to set cookies in the header, to base 64 encode the username and password for basic authentication, and so on.

Now we are ready to implement the body of the code:

```
  ops$tkyte@DEV816> create or replace package body http_pkg
    2  as
    3
    4  -- A global socket used by all of these routines.
    5  -- On any call, this socket will be OPENED and CLOSED
    6  -- before we return.
```

```
 7
 8  Socket       SocketType := SocketType(null);
 9
10  -- In support of the 7.3 and 8.0 original UTL_HTTP implementation,
11  -- REQUEST and REQUEST_PIECES are functionally equivalent
12  -- to these previous versions.
13  -- REQUEST and REQUEST_PIECES simple call the GET_URL
14  -- functions and convert their return types into the old
15  -- style return types.
16  -- REQUEST_PIECES has to do a moderate amount of very simple
17  -- parsing but overall, these functions are very straightforward.
18
19  function request( url      in varchar2,
20                    proxy in varchar2 default NULL )
21  return varchar2
22  is
23      l_content       clob;
24      l_status        number;
25      l_status_txt    varchar2(255);
26      l_httpHeaders   CorrelatedArray;
27      l_return        varchar2(2000);
28  begin
29      get_url( url, proxy, l_status, l_status_txt,
30               l_httpHeaders, l_content );
31
32      l_return := dbms_lob.substr( l_content, 2000, 1 );
33      dbms_lob.freetemporary( l_content );
34
35      return l_return;
36  end;
37
38  function request_pieces
39           (url in varchar2,
40            max_pieces natural default 32767,
41            proxy in varchar2 default NULL)
42  return html_pieces
43  is
44      l_content       clob;
45      l_status        number;
46      l_status_txt    varchar2(255);
47      l_httpHeaders   CorrelatedArray;
48      l_return        html_pieces;
49      l_offset        number default 1;
50      l_length        number;
51  begin
52      get_url( url, proxy, l_status, l_status_txt,
53               l_httpHeaders, l_content );
54
55      l_length := dbms_lob.getlength(l_content);
56      loop
57          exit when (l_offset > l_length OR
58                              l_return.count >= max_pieces );
59          l_return( l_return.count+1 ) :=
60              dbms_lob.substr(l_content,2000,l_offset );
61          l_offset := l_offset + 2000;
62      end loop;
63
64      dbms_lob.freetemporary( l_content );
```

```
 65
 66      return l_return;
 67  end;
 68
 69  -- A convenience routine to format and send the
 70  -- request to the HTTP server. We build the headers
 71  -- into a single string to be transmitted. The first
 72  -- line of the request is a GET or HEAD or POST followed
 73  -- by a URL and the protocol (we support only HTTP/1.0).
 74  --
 75  -- Then, we add to this string every other header record
 76  -- you asked us to send. This will contain cookies,
 77  -- authentication and so on.
 78  --
 79  -- If you are doing a POST, we add in the content length
 80  -- of the data we will post. The web server needs this
 81  -- so it knows where your request STOPS.
 82  --
 83  -- Lastly, we send the headers and if POST is the
 84  -- request type, the POST data itself.
 85
 86  procedure send_headers( p_url in varchar2,
 87                          p_httpHeaders in CorrelatedArray,
 88                          p_request in varchar2,
 89                          p_post_data in clob )
 90  is
 91      l_headers long;
 92  begin
 93      l_headers := p_request || ' ' ||
 94                   p_url || ' HTTP/1.0' ||
 95                   SocketType.crlf;
 96
 97      if ( p_httpHeaders is NOT null )
 98      then
 99          for i in 1 .. p_httpHeaders.vals.count
100          loop
101              l_headers := l_headers ||
102                           initcap(p_httpHeaders.vals(i).name) ||
103                           ': ' ||
104                           p_httpHeaders.vals(i).value ||
105                           SocketType.crlf;
106          end loop;
107      end if;
108      if ( p_request = 'POST' ) then
109          l_headers := l_headers || 'Content-length: ' ||
110                          dbms_lob.getlength(p_post_data) ||
111                         SocketType.crlf;
112          l_headers := l_headers ||
113             'Content-Type: application/x-www-form-urlencoded' ||
114              SocketType.crlf;
115      end if;
116      l_headers := l_headers || SocketType.crlf;
117      Socket.send( l_headers );
118
119      if ( p_request = 'POST' ) then
120          Socket.send( p_post_data );
121      end if;
122  end;
```

```
123
124 -- Another convenience routine. This takes the URL
125 -- you sent to us and figures out what server we need
126 -- to connect to, on what port and with what request...
127
128 procedure parse_server_port_url( p_url in varchar2,
129                                  p_proxy in varchar2,
130                                  p_server in out varchar2,
131                                  p_port    in out number,
132                                  p_url_out in out varchar2 )
133 is
134     l_colon number default instr( p_proxy, ':' );
135     l_url   long;
136     l_slash number;
137 begin
138     if ( p_proxy is NOT NULL )
139     then
140         if ( l_colon = 0 ) then
141             p_server := p_proxy;
142             p_port   := 80;
143         else
144             p_server := substr( p_proxy, 1, l_colon-1 );
145             p_port   := substr( p_proxy, l_colon+1 );
146         end if;
147         p_url_out :=  p_url;
148     else
149         if ( p_url not like 'http://%' ) then
150             raise init_failed;
151         end if;
152         l_url := substr( p_url, 8 );
153         l_slash := instr( l_url, '/' );
154         l_colon := instr( l_url, ':' );
155         if ( l_colon > 0 AND l_colon < l_slash )
156         then
157             p_server := substr( l_url, 1, l_colon-1 );
158             p_port   := substr( l_url, l_colon+1,
159                                        l_slash-(l_colon+1));
160             p_url_out:= substr( l_url, l_slash );
161         else
162             p_server := substr( l_url, 1, l_slash-1 );
163             p_port   := 80;
164             p_url_out:= substr( l_url, l_slash );
165         end if;
166     end if;
167 end;
168
169 -- These two routines are identical except one does a RAW
170 -- read and the other a TEXT read into a BLOB or CLOB.
171 -- They respect the content-length as set by the web server
172 -- if it is present, else they read to end of file
173 -- (signified by a NULL return from Socket.recv).
174
175 procedure get_clob_content( p_httpHeaders in CorrelatedArray,
176                             p_content     in out clob )
177 as
178     l_content_length number;
179     l_str_data         long;
180 begin
```

```
181      l_content_length :=
182             p_httpHeaders.valueof( 'Content-Length' );
183
184      dbms_lob.createtemporary( p_content, TRUE );
185      loop
186          exit when (dbms_lob.getlength(p_content) >=
187                                              l_content_length);
188          l_str_data := Socket.recv;
189          exit when ( l_str_data is NULL ); -- eof
190          dbms_lob.writeappend( p_content, length(l_str_data),
191                                l_str_data );
192      end loop;
193  end;
194
195  procedure get_blob_content( p_httpHeaders in CorrelatedArray,
196                              p_content     in out blob )
197  as
198      l_content_length number;
199      l_raw_data       long raw;
200  begin
201      l_content_length :=
202               p_httpHeaders.valueof( 'Content-Length' );
203
204      dbms_lob.createtemporary( p_content, TRUE );
205      loop
206          exit when (dbms_lob.getlength(p_content) >=
207                                            l_content_length);
208          l_raw_data := Socket.recv_raw;
209          exit when ( l_raw_data is NULL ); -- eof
210          dbms_lob.writeappend( p_content,
211                                utl_raw.length(l_raw_data),
212                                l_raw_data );
213      end loop;
214  end;
215
216  -- This internal GET_URL routine is used to connect to the
217  -- web server, send the request, and parse the HTTP headers
218  -- that come back. It does the 'real' work. The only
219  -- thing left after this is run is to read the actual content
220  -- of the web page.
221
222  procedure get_url_i( p_url         in      varchar2,
223                       p_proxy       in      varchar2,
224                       p_request     in      varchar2,
225                       p_status              out number,
226                       p_status_txt      out varchar2,
227                       p_httpHeaders in  out CorrelatedArray,
228                       p_post_data   in      clob default NULL )
229  is
230      l_server    varchar2(255);
231      l_port        number;
232      l_url        long;
233      l_str_data            long;
234  begin
235      parse_server_port_url( p_url, p_proxy, l_server,
236                             l_port, l_url );
237
238      Socket.initiate_connection( l_server, l_port );
```

```
239
240     send_headers( l_url, p_httpHeaders,
241                   p_request, p_post_data );
242
243     l_str_data := Socket.getline;
244     p_status := substr(l_str_data, instr(l_str_data,' ')+1, 3);
245     p_status_txt := l_str_data;
246
247     p_httpHeaders := CorrelatedArray(CorrelationArrayType());
248     loop
249         l_str_data := Socket.getline(true);
250         exit when l_str_data  is null or length(l_str_data) < 4;
251         p_httpHeaders.addHeaderline( l_str_data );
252    end loop;
253
254 end get_url_i;
255
256 -- The actual externalized procedures. These are what
257 -- we call from outside of this package. They simply
258 -- 'get' the URL and then get the content and close the
259 -- connection...
260 procedure get_url( p_url         in      varchar2,
261                    p_proxy       in      varchar2 default NULL,
262                    p_status             out number,
263                    p_status_txt      out varchar2,
264                    p_httpHeaders in  out CorrelatedArray,
265                    p_content     in  out clob )
266 is
267 begin
268     get_url_i( p_url, p_proxy, 'GET', p_status,
269                p_status_txt, p_httpHeaders );
270
271     get_clob_content( p_httpHeaders, p_content );
272     Socket.close_connection;
273 end get_url;
274
275 procedure get_url( p_url         in      varchar2,
276                    p_proxy       in      varchar2 default NULL,
277                    p_status          out number,
278                    p_status_txt      out varchar2,
279                    p_httpHeaders in  out CorrelatedArray,
280                    p_content     in  out blob )
281 is
282 begin
283     get_url_i( p_url, p_proxy, 'GET', p_status,
284                p_status_txt, p_httpHeaders );
285
286     get_blob_content( p_httpHeaders, p_content );
287     Socket.close_connection;
288 end get_url;
289
290 -- HEAD is even more simple than get above - no content
291 -- to retrieve.
292
293 procedure head_url( p_url        in     varchar2,
294                     p_proxy      in     varchar2 default null,
295                     p_status        out number,
296                     p_status_txt    out varchar2,
```

```
297                     p_httpHeaders   out CorrelatedArray )
298  is
299  begin
300      get_url_i( p_url, p_proxy, 'HEAD', p_status,
301                 p_status_txt, p_httpHeaders );
302      Socket.close_connection;
303  end;
304
305  -- A very simple routine that escapes all characters
306  -- found in your string that are in the l_bad string.
307  -- These characters cannot appear in the URL request
308  -- and cannot appear in posted data. It is illegal to
309  -- have a request such as http://host/foo?x=Hello%.
310  -- The % must be escaped so the string is:
311  -- http://host/foo?x=Hello%25
312
313  function urlencode( p_str in varchar2 ) return varchar2
314  as
315      l_tmp   varchar2(6000);
316      l_bad   varchar2(100) default ' >%}\~];?@&<#{|^[`/:=$+''"';
317      l_char  char(1);
318  begin
319      for i in 1 .. nvl(length(p_str),0) loop
320          l_char :=  substr(p_str,i,1);
321          if ( instr( l_bad, l_char ) > 0 )
322          then
323              l_tmp := l_tmp || '%' ||
324                               to_char( ascii(l_char), 'fmXX' );
325          else
326              l_tmp := l_tmp || l_char;
327          end if;
328      end loop;
329
330      return l_tmp;
331  end;
332
333  -- You may send up to 4 KB of cookie data. Cookies are
334  -- stashed in the header in the format:
335  -- name=value;name=value;name=value
336  --
337  -- As you can see, your cookie should not have = and ;
338  -- in it.
339
340  procedure add_a_cookie( p_name in varchar2,
341                          p_value in varchar2,
342                          p_httpHeaders in out CorrelatedArray )
343  is
344      l_cookie_string long;
345  begin
346      if ( p_httpHeaders is null ) then
347           p_httpHeaders :=
348              CorrelatedArray(CorrelationArrayType());
349      end if;
350
351      l_cookie_string := p_httpHeaders.valueof('Cookie') ||
352                          ';' || p_name || '=' || p_value;
353
354      p_httpHeaders.updatevalue( 'cookie',
```

```
355                                 ltrim(l_cookie_string,';') );
356  end;
357
358  -- To gain access to a page protected by basic
359  -- authentication, we must send an authorization header
360  -- record. The username/password must be base 64
361  -- encoded (obscured) before this happens. This
362  -- routine does that for us.
363
364  procedure set_basic_auth( p_username in varchar2,
365                            p_password in varchar2,
366                            p_httpHeaders in out CorrelatedArray )
367  is
368      l_b64encoded varchar2(255);
369  begin
370      if ( p_httpHeaders is null ) then
371           p_httpHeaders :=
372            CorrelatedArray(CorrelationArrayType());
373      end if;
374
375      simple_tcp_client.b64encode
376      ( utl_raw.cast_to_raw( p_username || ':' || p_password ),
377        l_b64encoded );
378
379      p_httpHeaders.updatevalue( 'Authorization',
380                                 'Basic ' || l_b64encoded );
381  end;
382
383  -- This routine lets you build a POST set of data. If
384  -- the amount of data you need to send to the web server is
385  -- larger, you should POST it. This expects you to start
386  -- with an uninitialized CLOB - we'll allocate space for it
387  -- and fill up the data. If you set p_urlencode to true,
388  -- this will escape all invalid characters automatically for
389  -- you...
390
391  procedure set_post_parameter
392            ( p_name in varchar2,
393              p_value in varchar2,
394              p_post_data in out clob,
395              p_urlencode in boolean default FALSE )
396  is
397      l_encoded_value        long;
398  begin
399      if ( p_post_data is null ) then
400          dbms_lob.createtemporary( p_post_data, TRUE );
401      else
402          dbms_lob.writeappend( p_post_data, 1, '&' );
403      end if;
404
405      dbms_lob.writeappend( p_post_data, length(p_name)+1,
406                            p_name||'=' );
407
408      if ( p_urlencode )
409      then
410          l_encoded_value := urlencode( p_value );
411      else
412          l_encoded_value := p_value;
```

```
413      end if;
414      dbms_lob.writeappend( p_post_data, length(l_encoded_value),
415                            l_encoded_value );
416  end;
417
418  -- The procedures for POSTing are very much the same as
419  -- for GETting except they take the data to POST as well.
420  -- They do the same exact logic otherwise...
421
422  procedure post_url(p_url        in     varchar2,
423                     p_post_data  in     clob,
424                     p_proxy      in     varchar2 default NULL,
425                     p_status        out number,
426                     p_status_txt    out varchar2,
427                     p_httpHeaders   in out CorrelatedArray,
428                     p_content    in out clob )
429  is
430  begin
431      get_url_i( p_url, p_proxy, 'POST', p_status, p_status_txt,
432                 p_httpHeaders, p_post_data );
433
434      get_clob_content( p_httpHeaders, p_content );
435      Socket.close_connection;
436  end ;
437
438  procedure post_url(p_url        in     varchar2,
439                     p_post_data  in     clob,
440                     p_proxy      in     varchar2 default NULL,
441                     p_status        out number,
442                     p_status_txt    out varchar2,
443                     p_httpHeaders   in out CorrelatedArray,
444                     p_content    in out blob )
445  is
446  begin
447      get_url_i( p_url, p_proxy, 'POST', p_status, p_status_txt,
448                 p_httpHeaders, p_post_data );
449
450      get_blob_content( p_httpHeaders, p_content );
451      Socket.close_connection;
452  end ;
453
454  end http_pkg;
455  /

Package body created.
```

In my opinion, the code is reasonably simple. The HTTP protocol is very straightforward – as many Internet protocols are. Now I would like to test the implementation of the above functions to ensure they work. This will also demonstrate the usage of many of the functions. In the following code, the procedure `P` is the `P` procedure I introduced in the `DBMS_OUTPUT` section to print long lines:

```
ops$tkyte@DEV816> create or replace procedure print_clob( p_clob in clob )
  2  as
  3       l_offset number default 1;
  4  begin
```

```
  5     loop
  6       exit when l_offset > dbms_lob.getlength(p_clob);
  7       dbms_output.put_line( dbms_lob.substr( p_clob, 255, l_offset ) );
  8       l_offset := l_offset + 255;
  9     end loop;
 10  end;
 11  /

Procedure created.

ops$tkyte@DEV816> create or replace
  2  procedure print_headers( p_httpHeaders correlatedArray )
  3  as
  4  begin
  5       for i in 1 .. p_httpHeaders.vals.count loop
  6           p( initcap( p_httpHeaders.vals(i).name ) || ': ' ||
  7                       p_httpHeaders.vals(i).value );
  8       end loop;
  9       p( chr(9) );
 10  end;
 11  /

Procedure created.
```

The above two procedures are convenience routines used in the test cases below. The `PRINT_CLOB` procedure will just print out 255 byte pieces of the `CLOB` for us. The `PRINT_HEADERS` routine takes and 'dumps' the `CorrelatedArray` type using the `P` procedure. This will enable us to easily see the HTTP headers that are returned. Now onto the test:

```
ops$tkyte@DEV816> begin
  2       p( http_pkg.request( 'http://aria/' ) );
  3  end;
  4  /
<HTML>
<HEAD>
<TITLE>Oracle Service Industries</TITLE>
</HEAD>
<FRAMESET COLS="130,*"
border=0>
<FRAME SRC="navtest.html" NAME="sidebar" frameborder=0>
<FRAME SRC="folder_home.html"
NAME="body" frameborder="0" marginheight="0" marginwidth="0">
</FRAMESET>

</BODY>
</HTML>

ops$tkyte@DEV816> declare
  2       pieces    http_pkg.html_pieces;
  3  begin
  4       pieces :=
  5           http_pkg.request_pieces( 'http://www.oracle.com',
  6                                    proxy=>'www-proxy1');
  7
  8       for i in 1 .. pieces.count loop
  9           p( pieces(i) );
 10       end loop;
```

```
 11  end;
 12  /
<head>
<title>Oracle Corporation</title>
<meta http-equiv="Content-Type" content="text/html;
...
```

The above two routines simply show that the `UTL_HTTP` methods of `REQUEST` and `REQUEST_PIECES` function as expected with our new package. Their functionality is identical. Now we will invoke our `URLENCODE` function that translates 'bad' characters into escape sequences in URLs and `POST` data:

```
ops$tkyte@DEV816> select
  2  http_pkg.urlencode( 'A>C%{hello}\fadfasdfads~`[abc]:=$+''"' )
  3  from dual;

HTTP_PKG.URLENCODE('A>C%{HELLO}\FADFASDFADS~`[ABC]:=$+''"')
------------------------------------------------------------
A%3EC%25%7Bhello%7D%5Cfadfasdfads%7E%60%5Babc%5D%3A%3D%24%2B%27%22
```

This shows that characters like > and % are escaped into `%3E` and `%25` respectively, and other sequences such as the word `hello` are not escaped. This allows us to use any of these special characters in our HTTP requests safely.

Now we will see the first of the new HTTP `URL` procedures. This procedure call will return Yahoo's homepage via a proxy server `www-proxy1` (you'll need your own proxy server of course; this is the proxy server behind my firewall). Additionally, we get to see the HTTP status returned – `200` indicates success. We also see the HTTP headers Yahoo returned to us. The mime-type will always be in there, and this tells us what type of content we can expect. Lastly, the content is returned and printed out:

```
ops$tkyte@DEV816> declare
  2       l_httpHeaders      correlatedArray;
  3       l_status     number;
  4       l_status_txt     varchar2(255);
  5       l_content      clob;
  6  begin
  7       http_pkg.get_url( 'http://www.yahoo.com/',
  8                         'www-proxy1',
  9                          l_status,
 10                          l_status_txt,
 11                          l_httpHeaders,
 12                          l_content );
 13
 14       p( 'The status was ' || l_status );
 15       p( 'The status text was ' || l_status_txt );
 16       print_headers( l_httpHeaders );
 17       print_clob( l_content );
 18  end;
 19  /
The status was 200
The status text was HTTP/1.0 200 OK

Date: Fri, 02 Feb 2001 19:13:26 GMT
Connection: close
```

```
Content-Type: text/html

<html><head><title>Yahoo!</title><base href=http://www.yahoo.com/><meta
http-equiv="PICS-Label"
```

Next, we will try the `HEAD` request against the home page of the Wrox web site and see what we can discover:

```
ops$tkyte@DEV816> declare
  2       l_httpHeaders      correlatedArray;
  3       l_status     number;
  4       l_status_txt     varchar2(255);
  5  begin
  6       http_pkg.head_url( 'http://www.wrox.com/',
  7                          'www-proxy1',
  8                           l_status,
  9                           l_status_txt,
 10                           l_httpHeaders );
 11
 12       p( 'The status was ' || l_status );
 13       p( 'The status text was ' || l_status_txt );
 14       print_headers( l_httpHeaders );
 15  end;
 16  /
The status was 200
The status text was HTTP/1.1 200 OK

Server: Microsoft-IIS/5.0
Date: Fri, 02 Feb 2001 19:13:26 GMT
Connection: Keep-Alive
Content-Length: 1270
Content-Type: text/html
Set-Cookie: ASPSESSIONIDQQQGGNQU=PNMNCIBACGKFLHGKLLBPEPMD; path=/
Cache-Control: private
```

From the headers, it is obvious that Wrox is running Windows with Microsoft IIS. Further, they are using ASP pages, as indicated by the cookie they sent back to us. If we were to have retrieved this page, it would have had 1,270 bytes of content.

Now we would like to see how cookies might work. Here, I am using a standard procedure that is used with OAS (Oracle Application Server), the `cookiejar` sample that shows how to use cookies in a PL/SQL web procedure. The routine `cookiejar` looks at the cookie value, and if set, increments it by one and returns it to the client. We'll see how that would work using our package. We are going to send the value 55 to the server so we are expecting it to send us 56 back:

```
ops$tkyte@DEV816> declare
  2       l_httpHeaders     correlatedArray;
  3       l_status     number;
  4       l_status_txt     varchar2(255);
  5       l_content     clob;
  6  begin
  7       http_pkg.add_a_cookie( 'COUNT', 55, l_httpHeaders );
  8       http_pkg.get_url
```

```
  9       ( 'http://aria.us.oracle.com/wa/webdemo/owa/cookiejar',
 10         null,
 11         l_status,
 12         l_status_txt,
 13         l_httpHeaders,
 14         l_content );
 15
 16       p( 'The status was ' || l_status );
 17       p( 'The status text was ' || l_status_txt );
 18       print_headers( l_httpHeaders );
 19       print_clob( l_content );
 20  end;
 21  /
The status was 200
The status text was HTTP/1.0 200 OK

Content-Type: text/html
Date: Fri, 02 Feb 2001 19:14:48 GMT
Allow: GET, HEAD
Server: Oracle_Web_listener2.1/1.20in2
Set-Cookie: COUNT=56; expires=Saturday, 03-Feb-2001 22:14:48 GMT

<HTML>
<HEAD>
<TITLE>C is for Cookie</TITLE>
</HEAD>
<BODY>
<HR>
<IMG SRC="/ows-img/ows.gif">
<H1>C
is for Cookie</H1>
<HR>
You have visited this page <STRONG>56</STRONG> times in the last 24
hours.
...
```

As you can see, the cookie value of 55 was transmitted and the server incremented it to 56. It then sent us back the modified value along with an expiration date.

Next, we would like to see how to access a page that requires a username and password. This is done via the following:

```
ops$tkyte@DEV816> declare
  2      l_httpHeaders   correlatedArray;
  3      l_status        number;
  4      l_status_txt    varchar2(255);
  5      l_content       clob;
  6  begin
  7      http_pkg.set_basic_auth( 'tkyte', 'tiger', l_httpheaders );
  8      http_pkg.get_url
  9      ( 'http://aria.us.oracle.com:80/wa/intranets/owa/print_user',
 10        null,
 11        l_status,
 12        l_status_txt,
 13        l_httpHeaders,
 14        l_content );
```

```
 15
 16      p( 'The status was ' || l_status );
 17      p( 'The status text was ' || l_status_txt );
 18      print_headers( l_httpHeaders );
 19      print_clob(l_content);
 20  end;
 21  /
The status was 200
The status text was HTTP/1.0 200 OK

Content-Type: text/html
Date: Fri, 02 Feb 2001 19:49:17 GMT
Allow: GET, HEAD
Server: Oracle_Web_listener2.1/1.20in2

remote user = tkyte
```

Here, I just set up a DAD (database access descriptor, an OAS term) that did not store the username/password with the DAD. This means the web server is expecting the request to contain the username/password to use. Here, I passed my credentials to a routine that simply printed out the REMOTE_USER cgi-environment variable in PL/SQL (the name of the remotely connected user).

Lastly, we would like to demonstrate the POSTing of data. Here I am using a URL from Yahoo again. Yahoo makes it easy to get stock quotes in a spreadsheet format. Since the list of stock symbols you might be interested in could get quite large, we would suggest POSTing this data. Here is an example that gets a couple of stock quotes/indices from Yahoo using HTTP. The data will be returned in CSV (comma-separated values) for easy parsing and loading into a table for example:

```
ops$tkyte@DEV816> declare
  2      l_httpHeaders   correlatedArray;
  3      l_status        number;
  4      l_status_txt    varchar2(255);
  5      l_content       clob;
  6      l_post          clob;
  7  begin
  8      http_pkg.set_post_parameter( 'symbols','orcl ^IXID ^DJI ^SPC',
  9                                   l_post, TRUE );
 10      http_pkg.set_post_parameter( 'format', 'sl1d1t1c1ohgv',
 11                                   l_post, TRUE );
 12      http_pkg.set_post_parameter( 'ext', '.csv',
 13                                   l_post, TRUE );
 14      http_pkg.post_url( 'http://quote.yahoo.com/download/quotes.csv',
 15                          l_post,
 16                         'www-proxy',
 17                          l_status,
 18                          l_status_txt,
 19                          l_httpHeaders,
 20                          l_content );
 21
 22      p( 'The status was ' || l_status );
 23      p( 'The status text was ' || l_status_txt );
 24      print_headers( l_httpHeaders );
```

```
 25      print_clob( l_content );
 26  end;
 27  /
The status was 200
The status text was HTTP/1.0 200 OK

Date: Fri, 02 Feb 2001 19:49:18 GMT
Cache-Control: private
Connection: close
Content-Type: application/octet-stream

"ORCL",28.1875,"2/2/2001","2:34PM",-1.875,29.9375,30.0625,28.0625,26479100
"^IXID",1620.60,"2/2/2001","2:49PM",-45.21,1664.55,1667.46,1620.40,N/A
"^DJI",10899.33,"2/2/2001","2:49PM",-84.30,10982.71,11022.78,10888.01,N/A
"^SPC",1355.17,"2/2/2001","2:49PM",-18.30,1373.53,1376.16,1354.21,N/A
```