|  |
| --- |
| **Oracle UTL\_HTTP Package**  *Oracle Database Tips by Donald Burleson* |

[*Advanced Oracle Utilities: The Definitive Reference*](http://www.rampant-books.com/book_0801_oracle_utilities.htm)*by Rampant TechPress is written by top Oracle database experts (Bert Scalzo, Donald Burleson, and Steve Callan).  The following is an excerpt from the book.*

UTL\_HTTP contains two functions; specifically, REQUEST and REQUEST\_LINES. REQUEST is a function that returns a string variable (4000 character maximum). REQUEST will return a string that contains up to the first 4000 bytes of the HTML result returned from the HTTP request to a specific URL. In other words, if more than 4000 characters are in the resulting HTML received, characters 4001 and beyond will be truncated.

According to the package specification: Package UTL\_HTTP contains function REQUEST for making HTTP callouts from PL/SQL programs. Function UTL\_HTTP.REQUEST may not be called directly from an SQL statement (because its return type is not supported in a query); however, it may be called from the body of another function that is called directly from an SQL statement (and that returns a type that is legal for a function called from SQL; that is, CHAR).

The input parameters to the REQUEST function are URL and PROXY. URL is a varchar2 string that will contain the URL to be called. PROXY contains the IP address or domain name for the proxy server; the default is NULL (meaning there is no proxy server).

The REQUEST\_PIECES function also calls a URL and can accept a PROXY server value. However, REQUEST\_PIECES returns a PL/SQL table (array structure) of varchar2 strings containing the entire site's HTML output (or at least up to 32,767 times 4000 characters of the page).

According to the UTL\_HTTP specification, the function REQUEST\_LINES takes a URL as its argument and returns a PL/SQL-table of strings, which are the successive pieces of the HTML response obtained from the HTTP request to that URL.

REQUEST\_LINES accepts three parameters: URL, MAX\_PIECES, and PROXY. URL and PROXY are described above. MAX\_PIECES represents the maximum number of "pieces" to return. The default value is 32,767.

The UTL\_HTTP package also contains a data-type of HTML\_PIECES, which is a varchar2 PL/SQL table. This datatype should be used in conjunction with the REQUEST\_LINES function.

REQUEST and REQUEST\_LINES are typically just called with a URL (unless you have a proxy server) as follows:

REQUEST

declare html\_results

varchar2(4000); -- max is 2000 for v7

begin

html\_results := utl\_http.request

('http://www.tusc.com');

.... -- process html\_results

end;

REQUEST\_LINES

declare pieces

utl\_http.html\_pieces;

begin

pieces := utl\_http.request\_

lines('http://www.tusc.com');

for i in 1 .. pieces.count

loop

... -- process each piece

... -- refer to each piece as pieces(i)

end loop;

      end;

Some things worth noting include:

1. Although it seems repetitious, http:// (the protocol) is specified in the URL.

2. Note the use of the PL/SQL-table method COUNT to discover the number of pieces returned, which may be zero or more.

3. When using REQUEST\_LINES, MAX\_PIECES is optional. It is the maximum number of pieces (each 4000 characters in length except for the last, which may be shorter) that REQUEST should return. If provided, that argument should be a positive integer.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In terms of scale, the documentation behind the UTL\_HTTP package is monstrous compared to many other packages. Given the huge amount of data under the covers of a Web page, it is not surprising that the number of subprograms for this package would be quite numerous.

What is a quick and dirty use of UTL\_HTTP? The simplest and fastest example is one's ability to capture the source of a Web page by using PL/SQL. Coupled with spooling the output, one now has a simple means of capturing the source of virtually any Web page.

If the developer can spool to a file, she can load the file into the database. If there is a BLOB in the database, write it out to the file system. This workflow should bring to mind the idea of creating a rudimentary source control system. Not only can the developer store and generate HTML files based on a combination of UTL\_FILE and UTL\_HTTP, but they can be edited as well.

There are applications where the HTML for Web pages served in a framework is stored in the database. How those pages render is slightly different than creating HTML files on the file system, but the concept is the same - the database serves up the source for a page. So look at a quick example of capturing the source for a page. Use the home page of dba-oracle.com. This small body of code will generate almost 900 lines of HTML code and uses eight subprograms in the UTL\_HTTP package.

spool get\_page.html

DECLARE

 req   UTL\_HTTP.REQ;

 resp  UTL\_HTTP.RESP;

 value VARCHAR2(1024);

BEGIN

  req := UTL\_HTTP.BEGIN\_REQUEST('http://dba-oracle.com');

  UTL\_HTTP.SET\_HEADER(req, 'User-Agent', 'Mozilla/4.0');

  resp := UTL\_HTTP.GET\_RESPONSE(req);

  LOOP

    UTL\_HTTP.READ\_LINE(resp, value, TRUE);

    dbms\_output.put\_line(value);

  END LOOP;

  UTL\_HTTP.END\_RESPONSE(resp);

EXCEPTION

  WHEN UTL\_HTTP.END\_OF\_BODY THEN

    UTL\_HTTP.END\_RESPONSE(resp);

END;

/

spool off

**Here is the top part of the captured HTML source.**

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">

<html>

<head>

<META name="verify-v1" content="03pBD3fe1Hr9cZGVzdKBKWKwK7myXtC2l7tXPLFZzbI=" />

<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">

<meta name="GENERATOR" content="Microsoft FrontPage 6.0">

<title>Oracle Consulting, Oracle Support and Oracle Training by BC</title>

<meta name="keywords" content="Oracle Consulting, Oracle support, Oracle Consultants,Oracle Education,Oracle

contracting,Oracle consulting,Oracle consultant,Oracle Training,Oracle dba support,Oracle architecture,Oracle

classes,Oracle,Oracle architecture,Oracle tuning, Oracle data warehousing">

<meta name="description" content="Burleson Oracle consulting, Oracle training and Oracle support">

</head>

Any Web page that can be accessed via the browser, minus those requiring authentication, can be captured. With this ability, one can sample pages and check to see if there are any differences. But what about HTTPS pages? For these pages, one needs to configure Wallet Manager.

On Windows, this can be found under Start > Programs > the Oracle home > Integrated Management Tools > Wallet Manager, and in UNIX, use one found in $ORACLE\_HOME/bin. Use of Wallet Manager enables the developer to capture secure Web pages, secure in the sense that this is what one is allowed to access via the wallet information.

If the returned page is large, use the REQUEST\_PIECES function. This function returns a PL/SQL table of 2000 bytes. In the prior example, how many pieces were returned and what was the length? Using the example in Oracle's documentation, and correcting the error for the length output (use "len," not "i" in the last DBMS\_OUTPUT), the following occurs:

DECLARE

   x   UTL\_HTTP.HTML\_PIECES;

   len PLS\_INTEGER;

BEGIN

   x := UTL\_HTTP.REQUEST\_PIECES('http://dba-oracle.com/', 100);

   DBMS\_OUTPUT.PUT\_LINE(x.count || ' pieces were retrieved.');

   DBMS\_OUTPUT.PUT\_LINE('with total length ');

   IF x.count < 1 THEN

      DBMS\_OUTPUT.PUT\_LINE('0');

  ELSE

   len := 0;

   FOR i in 1..x.count LOOP

      len := len + length(x(i));

   END LOOP;

   DBMS\_OUTPUT.PUT\_LINE(len);

  END IF;

END;

/

23 pieces were retrieved.

with total length

44356

The output using this against www.dba-oracle.com is 22 pieces with length 43086, which is quite different than what is shown in Oracle's documentation (4 pieces and length 7687).

-------------------------------------------------------------------------

This UTL\_HTTP (Hyper Text Transfer Protocol) package is available from the Oracle version 9.2 for accessing data on the internet using HTTP. This package also fetches the internet data using HTTP over Secured Socket Layer (SSL), also known as HTTPS. For the HTTPS fetches, the SSL client authentication has to be performed by sending the client certificate in a wallet to the remote website.

The different objects available in this package are described below.

**REQ Type**

This type is used for representing an HTTP request. This type accepts the information returned by the BEGIN\_REQUEST function in READ ONLY mode. Thus, changing the type value does not have any impact on the request.

The prototype of the REQ type is shown below,

TYPE req IS RECORD (

   url           VARCHAR2(32767),

   method        VARCHAR2(64),

   http\_version  VARCHAR2(64));

·         **URL**parameter holds the URL of the HTTP request.

·         **METHOD**parameter holds the method to be used on the resource identified by the HTTP request.

·         **HTTP\_VERSION**parameter holds the HTTP version used during the HTTP request.

|  |
| --- |
| % Note: All the above parameters are set only after the request is created using the BEGIN\_REQUEST function. |

**BEGIN\_REQUEST Function**

This function begins a new HTTP request by establishing a network connection between the PL/SQL program and the target website meanwhile sending the request line. This function’s return value is set to the REQ type for continuing the request.

The prototype of the BEGIN\_REQUEST function is shown below,

UTL\_HTTP.BEGIN\_REQUEST (

   url           IN VARCHAR2,

   method        IN VARCHAR2 DEFAULT 'GET',

   http\_version  IN VARCHAR2 DEFAULT NULL)

RETURN req;

**UTL\_HTTP RESP Type**

This type is used for representing an HTTP response. This type accepts the information returned by the GET\_REQUEST function in READ ONLY mode. Thus, changing the type value does not have any impact on the response.

TYPE resp IS RECORD (

   status\_code    PLS\_INTEGER,

   reason\_phrase  VARCHAR2(256),

   http\_version   VARCHAR2(64));

·         **STATUS\_CODE**parameter accepts a three-digit status code indicating the results of the HTTP request.

·         **REASON\_PHRASE**parameter accepts the short message describing the status code returned by the HTTP request.

|  |
| --- |
| % Note: All these parameters are set after the response is processed by the GET\_RESPONSE type. |

**UTL\_HTTP GET\_RESPONSE Function**

This function reads the HTTP response accepting the request as its input. The status line and the response headers are read right before the function’s return into the RESP type, completing the header section.

The prototype of the GET\_RESPONSE function is shown below,

UTL\_HTTP.GET\_RESPONSE (

   r IN OUT NOCOPY req)

RETURN resp;

·         **R**parameter denotes the HTTP response.

**UTL\_HTTP SET\_HEADER Procedure**

This procedure is used for setting the header section for an HTTP request. We can also set multiple headers of the same name for a request as the duplicate header name will not replace the existing headers of the same name.

|  |
| --- |
| % Note: If the request is made using the HTTP 1.1 version, the header is automatically set. |

The prototype for defining the SET\_HEADER procedure is shown below,

UTL\_HTTP.SET\_HEADER (

   r IN OUT NOCOPY req,

   name IN VARCHAR2,

   value IN VARCHAR2);

·         **R**parameter accepts the HTTP request made.

·         **NAME**parameter accepts a user-defined name in the request header.

·         **VALUE**parameter accepts a user-defined value for the request header.

In the below example, the BEGIN\_REQUEST function accepts the URL of GOOGLE website and begins the HTTP request which is then assigned to the REQ type. This REQ type variable is then passed to the SET\_HEADER procedure for setting a user-defined header. After the header is added, the response for the request is received and stored in the RESP type using the GET\_RESPONSE function. The GET\_HEADER\_COUNT function returns the number of HTTP response headers returned with the response. It is then looped for the number of times to the GET\_HEADER function to retrieve the header name and its value.

After retrieving the response, it is then looped using the READ\_LINE procedure and buffers it using the PUT\_LINE procedure. The READ\_LINE procedure accepts the response type variable and the output variable which holds the HTTP response body in the text format. The third Boolean parameter omits the newline character when setting to *true* or is included in case of *false*.

Once the response body reaches its end, the END\_OF\_BODY exception is thrown, which is handled separately to avoid sudden termination of the program. After the response body is buffered out, the request and the response is completed by using the END\_RESPONSE procedure.

DECLARE

  l\_http\_req UTL\_HTTP.REQ;

  l\_http\_resp UTL\_HTTP.RESP;

  l\_vc\_header\_name varchar2(256);

  l\_vc\_header\_value varchar2(1024);

  l\_vc\_html VARCHAR2(32767);

BEGIN

  l\_http\_req := UTL\_HTTP.BEGIN\_REQUEST('http://www.google.com');

  dbms\_output.put\_line('Request URL: '||l\_http\_req.url);

  dbms\_output.put\_line('Request Method: '||l\_http\_req.method);

  dbms\_output.put\_line('Request Version: '||l\_http\_req.http\_version);

  UTL\_HTTP.SET\_HEADER(l\_http\_req, 'Header #1', 'Chrome V.52.X');

  l\_http\_resp := UTL\_HTTP.GET\_RESPONSE(l\_http\_req);

  dbms\_output.put\_line('Response Status Code: '||l\_http\_resp.status\_code);

  dbms\_output.put\_line('Response Reason: '||l\_http\_resp.reason\_phrase);

  dbms\_output.put\_line('Response Version: '||l\_http\_resp.http\_version);

  dbms\_output.put\_line('---Header Count Starts---');

  FOR loop\_hc IN 1..UTL\_HTTP.GET\_HEADER\_COUNT(l\_http\_resp)

  LOOP

    UTL\_HTTP.GET\_HEADER(l\_http\_resp, loop\_hc, l\_vc\_header\_name, l\_vc\_header\_value);

    DBMS\_OUTPUT.PUT\_LINE(l\_vc\_header\_name || ': ' || l\_vc\_header\_value);

  END LOOP loop\_hc;

  dbms\_output.put\_line('---Header Count Ends---');

  LOOP

    UTL\_HTTP.READ\_LINE(l\_http\_resp, l\_vc\_html, TRUE);

    DBMS\_OUTPUT.PUT\_LINE(l\_vc\_html);

  END LOOP;

  UTL\_HTTP.END\_RESPONSE(l\_http\_resp);

EXCEPTION

WHEN UTL\_HTTP.END\_OF\_BODY THEN

  UTL\_HTTP.END\_RESPONSE(l\_http\_resp);

END;

/

**Result:**

Request URL: http://www.google.com

Request Method: GET

Request Version:

Response Status Code: 200

Response Reason: OK

Response Version: HTTP/1.1

---Header Count Starts---

Date: Mon, 05 Sep 2016 17:39:14 GMT

Expires: -1

Cache-Control: private, max-age=0

Content-Type: text/html; charset=ISO-8859-1

Server: gws

X-XSS-Protection: 1; mode=block

---Header Count Ends---

<HTML File for the GOOGLE webpage>

X-Frame-Options: SAMEORIGIN

Accept-Ranges: none

Vary: Accept-Encoding

Connection: close

**HTTP Cookies**

HTTP cookies are a small amount of data sent by the website and stored in the user’s browser while the user is browsing. These are originally designed for remembering the basic state of the user’s browsing activity such as pages visited, items added to the cart while e-shopping, form fields fill-up, etc. The term is coined from the *fortune cookies*, a cookie with an embedded message.

The COOKIE record-type represents an HTTP cookie and the COOKIE\_TABLE index by table represents a set of HTTP cookies.

The prototype of the COOKIE and the COOKIE\_TABLE type is shown below,

TYPE cookie IS RECORD (

   name  VARCHAR2(256),

   value  VARCHAR2(1024),

   domain  VARCHAR2(256),

   expire  TIMESTAMP WITH TIME ZONE,

   path  VARCHAR2(1024),

   secure  BOOLEAN,

   version  PLS\_INTEGER,

   comment  VARCHAR2(1024));

TYPE cookie\_table IS TABLE OF cookie INDEX BY binary\_integer;

·         **NAME**parameter holds the name of the cookie.

·         **VALUE**parameter holds the value of the cookie.

·         **DOMAIN**parameter holds the domain for which the cookie is valid.

·         **EXPIRE**parameter holds the expiry time for the cookie.

·         **PATH**parameter holds the subset of URLs for which the cookie is available.

·         **SECURE**parameter holds the Boolean for whether the cookie should be returned securely to the web server or not.

·         **VERSION**parameter holds the version of the HTTP cookie specification.

·         **COMMENT**parameter holds the description about the HTTP cookie.

These cookies are maintained by the UTL\_HTTP package transparently lasting for the duration of the session and are not usually changed by the PL/SQL programs. When we want to maintain the cookies beyond their lifetime, we must download them into the database tables using the GET\_COOKIES procedure and use them for the next database session using the ADD\_COOKIES procedure. We must capture all the information on the cookie (Except its comment) mandatorily for the cookie to function properly. We must also ensure that we do not change the cookie information as it may result in application failure.

**GET\_COOKIES Procedure**

This procedure returns all the cookies currently maintained by the UTL\_HTTP package set by all the web servers.

The prototype for defining the GET\_COOKIES procedure is shown below,

UTL\_HTTP.GET\_COOKIES (cookies IN OUT NOCOPY cookie\_table);

·         **COOKIES**parameter returns all the cookies available.

**ADD\_COOKIES Procedure**

This procedure adds all the cookies that are currently maintained by the UTL\_HTTP package.

The prototype for defining the ADD\_COOKIES procedure is shown below,

UTL\_HTTP.ADD\_COOKIES (cookies IN cookie\_table);

·         **COOKIES**parameter adds all the cookies available.

In the below example, cookies from one session have been preserved and used for another session. Firstly, the MY\_COOKIE\_TABLE with the attributes of the type UTL\_HTTP.COOKIE\_TABLE is created as shown below. Note that the COMMENT attribute is not included as it is not mandatory.

CREATE

  TABLE my\_cookie\_table

  (

    name    VARCHAR2(256),

    value   VARCHAR2(1024),

    domain  VARCHAR2(256),

    expire  DATE,

    path    VARCHAR2(1024),

    secure  VARCHAR2(1),

    version INTEGER

  );

Then, the cookies from one session are copied into this table using the below anonymous block. In the below example, the cookies transparently stored in the UTL\_HTTP package are retrieved using the GET\_COOKIES procedure and is assigned to the COOKIE\_TABLE type variable. It is then looped for every cookie value and then inserted into our MY\_COOKIE\_TABLE permanently. Note that the attribute SECURE is converted to Y for *true* and N for *false values* of the cookie information.

DECLARE

  l\_ct\_cookies UTL\_HTTP.COOKIE\_TABLE;

  l\_vc\_secure VARCHAR2(1);

BEGIN

  UTL\_HTTP.GET\_COOKIES(l\_ct\_cookies);

  FOR loop\_ct IN 1..l\_ct\_cookies.count

  LOOP

    IF (l\_ct\_cookies(loop\_ct).secure) THEN

      l\_vc\_secure := 'Y';

    ELSE

      l\_vc\_secure := 'N';

    END IF;

    INSERT

    INTO

      my\_cookie\_table VALUES

      (

        l\_ct\_cookies(loop\_ct).name,

        l\_ct\_cookies(loop\_ct).value,

        l\_ct\_cookies(loop\_ct).domain,

        l\_ct\_cookies(loop\_ct).expire,

        l\_ct\_cookies(loop\_ct).path,

        l\_vc\_secure,

        l\_ct\_cookies(loop\_ct).version

      );

  END LOOP loop\_ct;

  COMMIT;

END;

/

When we query the MY\_COOKIE\_TABLE after executing the above anonymous block, we are able to see around 5 cookies being downloaded and readily available for adding to any other session.

SELECT \* FROM my\_cookie\_table;

**Query Result:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **NAME** | **VALUE** | **DOMAIN** | **EXPIRE** | **PATH** | **SECURE** | **VERSION** |
| GUEST\_LANGUAGE\_ID | en\_US | [www.peteranswers.com](http://www.peteranswers.com/) | 05-SEP-2017 | / | N |  |
| JSESSIONID | 324DF1EE44890ED6  3ADB92FE98E1FF85 | [www.peteranswers.com](http://www.peteranswers.com/) |  | / | Y |  |
| COOKIE\_SUPPORT | True | [www.peteranswers.com](http://www.peteranswers.com/) | 05-SEP-2017 | / | N |  |
| NID | 85=D6HVRmZc3nuMGyzRQZ8  MCZw5iLNht16AfAnmeRSu  8jOR73q4DVlnMlMAMBkji0kjL  BZ36jITf8ZVKm0WLW7JGd1-9LFX9nOFVEOZxos1x4  BTA04pONU\_wONMWajuYXAl | .google.co.in | 07-MAR-2017 | / | N |  |
| JSESSIONID | 2515B7F1D123CD7C9  5597CF5A953EC00 | 100pulse.com |  | / | N |  |

After copying the cookies, we can add them to any session by using the below anonymous block. The MY\_COOKIE\_TABLE with all the cookies is copied into a COOKIE\_TABLE type variable and is called as input for the ADD\_COOKIES procedure. Before that, we can either delete all the existing cookies using the CLEAR\_COOKIES procedure or add them up to the existing cookies. The total cookies available in the UTL\_HTTP package state can be retrieved using the GET\_COOKIE\_COUNT function.

DECLARE

  l\_ct\_cookies UTL\_HTTP.COOKIE\_TABLE;

  l\_c\_cookie UTL\_HTTP.COOKIE;

  l\_pi\_counter PLS\_INTEGER := 0;

  CURSOR cur

  IS

    SELECT

      \*

    FROM

      my\_cookie\_table;

BEGIN

  FOR loop\_cur IN cur

  LOOP

    l\_pi\_counter        :=l\_pi\_counter+1;

    l\_c\_cookie.name     := loop\_cur.name;

    l\_c\_cookie.value    := loop\_cur.value;

    l\_c\_cookie.domain   := loop\_cur.domain;

    l\_c\_cookie.expire   := loop\_cur.expire;

    l\_c\_cookie.path     := loop\_cur.path;

    IF (loop\_cur.secure  = 'Y') THEN

      l\_c\_cookie.secure := TRUE;

    ELSE

      l\_c\_cookie.secure := FALSE;

    END IF;

    l\_c\_cookie.version         := loop\_cur.version;

    l\_ct\_cookies(l\_pi\_counter) := l\_c\_cookie;

  END LOOP loop\_cur;

  UTL\_HTTP.CLEAR\_COOKIES;

  UTL\_HTTP.ADD\_COOKIES(l\_ct\_cookies);

  dbms\_output.put\_line('Total Cookies Count: '||UTL\_HTTP.GET\_COOKIE\_COUNT);

END;

/

**Result:**

Total Cookie Count: 5

**REQUEST\_CONTEXT\_KEY Subtype**

This subtype creates a unique PLS\_INTEGER key for representing a request context. A request context is a context that holds wallet information and a cookie table for private HTTP request and response. This wallet and the cookie table will not be shared with any other application in the same database session.

The prototype of this type is shown below,

SUBTYPE request\_context\_key IS PLS\_INTEGER;

**CREATE\_REQUEST\_CONTEXT Function**

This function is used for creating a request context. A request context holds a wallet information and a cookie table for making a private HTTP request without sharing it with any other application in the same database session.

The prototype for defining this function is shown below,

UTL\_HTTP.CREATE\_REQUEST\_CONTEXT (

         wallet\_path IN VARCHAR2 DEFAULT NULL,

         wallet\_password IN VARCHAR2 DEFAULT NULL,

         enable\_cookies  IN BOOLEAN DEFAULT TRUE,

         max\_cookies IN PLS\_INTEGER DEFAULT 300,

         max\_cookies\_per\_site IN PLS\_INTEGER DEFAULT 20)

         RETURN request\_context\_key;

·         **WALLET\_PATH**parameter accepts the directory path containing the wallet.

·         **WALLET\_PASSWORD**parameter accepts the password needed for opening the wallet.

·         **ENABLE\_COOKIES**parameter is used for enabling (Set True) and disabling (Set False) the cookie support for the HTTP request.

·         **MAX\_COOKIES** parameter sets the maximum cookie limit to be maintained during this HTTP request context.

·         **MAX\_COOKIES\_PER\_SITE**parameter sets the maximum cookie limit to be maintained for each site during this HTTP request context.

**DESTROY\_REQUEST\_CONTEXT Procedure**

This procedure is used for destroying a request context. Note that a request context cannot be destroyed when it is in use by an HTTP request or a response.

The prototype for defining this procedure is shown below,

UTL\_HTTP.DESTROY\_REQUEST\_CONTEXT (request\_context request\_context\_key);

·         **Request\_context**parameter accepts a request context that has to be destroyed.

In the below example, a request context is created with wallet and cookie information. After the context creation, an HTTP request and its corresponding response are triggered for the user passed web URL which we have discussed in our previous sections. After the HTML file is retrieved, the request context is destroyed.

To know more about the wallet creation and its functioning, please refer to the *Transparent Data Encryption* section in *Chapter 8: The Advanced Security Methods in PL/SQL*.

DECLARE

  l\_vc\_url VARCHAR2(100):='www.anexampleurl.com';

  l\_rc\_key UTL\_HTTP.REQUEST\_CONTEXT\_KEY;

  l\_http\_req UTL\_HTTP.REQ;

  l\_http\_resp UTL\_HTTP.RESP;

  l\_vc\_html VARCHAR2(1024);

BEGIN

  l\_rc\_key := UTL\_HTTP.CREATE\_REQUEST\_CONTEXT( wallet\_path =>

  'File:C:\app\BoobalGanesan\admin\oracle\wallet', wallet\_password => 'oracle',

  enable\_cookies => TRUE, max\_cookies => 100, max\_cookies\_per\_site => 10);

  dbms\_output.put\_line('Request Context Key: '||l\_rc\_key);

  l\_http\_req  := UTL\_HTTP.BEGIN\_REQUEST(l\_vc\_url,l\_rc\_key);

  l\_http\_resp := UTL\_HTTP.GET\_RESPONSE(l\_http\_req);

  BEGIN

    LOOP

      UTL\_HTTP.READ\_LINE(l\_http\_resp, l\_vc\_html,true);

      DBMS\_OUTPUT.PUT\_LINE(l\_vc\_html);

    END LOOP;

  EXCEPTION

  WHEN UTL\_HTTP.END\_OF\_BODY THEN

    UTL\_HTTP.END\_RESPONSE(l\_http\_resp);

  END;

  UTL\_HTTP.DESTROY\_REQUEST\_CONTEXT(l\_rc\_key);

END;

/

**Result:**

Request Context Key: 1471220475

<HTML File for the webpage>

**Simple HTTP fetches**

The functions REQUEST and REQUEST\_PIECES accept a URL as its parameter and contacts the website for returning the data obtained in HTML format.

**REQUEST Function**

This function returns the information up to the first 2000 bytes obtained from the input URL. This function can be called in a SELECT statement for a quick call out of small websites.

The prototype for defining this function is shown below,

UTL\_HTTP.REQUEST (

         url IN VARCHAR2,

         proxy IN VARCHAR2 DEFAULT NULL,

         wallet\_path IN VARCHAR2 DEFAULT NULL,

         wallet\_password IN VARCHAR2 DEFAULT NULL)

         RETURN VARCHAR2;

·         **URL**parameter accepts an input URL for the data fetch.

·         **PROXY**parameter accepts the proxy server to use during an HTTP request.

·         **WALLET\_PATH**parameter accepts the directory path for the wallet.

·         **WALLET\_PASSWORD**parameter accepts the password for opening the wallet.

In the below example, a URL is passed as input to the REQUEST function returning the first 2000 bytes of information obtained from the website.

SELECT utl\_http.request('http://www.dba-oracle.com/') FROM dual;

**Result:**

<2000 bytes of the HTML file are obtained from the input URL>

**REQUEST\_PIECES Function**

This function returns the complete information of the input URL in the form of PL/SQL associative array having a maximum of 2000 byte. This column has an extra attribute compared to the REQUEST function for passing the maximum pieces allowed during the HTML file split up.

This function cannot be used in a SELECT statement as it returns an associative array. Also, the TABLE function cannot be used upon this function’s associative array return statement w.r.t. the 12c enhancement as the MAX\_PIECES parameter uses a non-SQL datatype.

The prototype of the HTML\_PIECES type and the REQUEST\_PIECES function is shown below,

TYPE html\_pieces IS TABLE OF VARCHAR2(2000) INDEX BY BINARY\_INTEGER;

UTL\_HTTP.REQUEST\_PIECES(

         url IN VARCHAR2,

         max\_pieces IN NATURAL DEFAULT 32767,

         proxy IN VARCHAR2 DEFAULT NULL,

         wallet\_path IN VARCHAR2 DEFAULT NULL,

         wallet\_password IN VARCHAR2 DEFAULT NULL)

         RETURN html\_pieces;

·         **MAX\_PIECES**parameter accepts a positive integer value for determining the maximum number of chunks the obtained HTML must be split into.

In the below example, the REQUEST\_PIECES function requests for a maximum of 100 chunks of 2000-bytes information from the input URL.

This function’s return statement is then looped to print all the available chunks of 2000 bytes (The last chunk may contain <=2000 bytes).

DECLARE

  l\_html\_pieces utl\_http.html\_pieces;

  LEN PLS\_INTEGER;

BEGIN

  l\_html\_pieces:=UTL\_HTTP.REQUEST\_PIECES('http://www.google.com', 100);

  dbms\_output.put\_line('Total chunks: '||l\_html\_pieces.count);

  FOR loop\_chunks IN 1..l\_html\_pieces.count

  LOOP

    dbms\_output.put\_line('Chunk #'||loop\_chunks||': '||l\_html\_pieces(loop\_chunks));

  END LOOP loop\_chunks;

END;

/

**Result:**

Total chunks: 24

Chunk #1: <HTML file of 2000 bytes>

Chunk #2: <HTML file of 2000 bytes>

Chunk #3: <HTML file of 2000 bytes>

...

...

Chunk #24: <HTML file of <=2000 bytes>