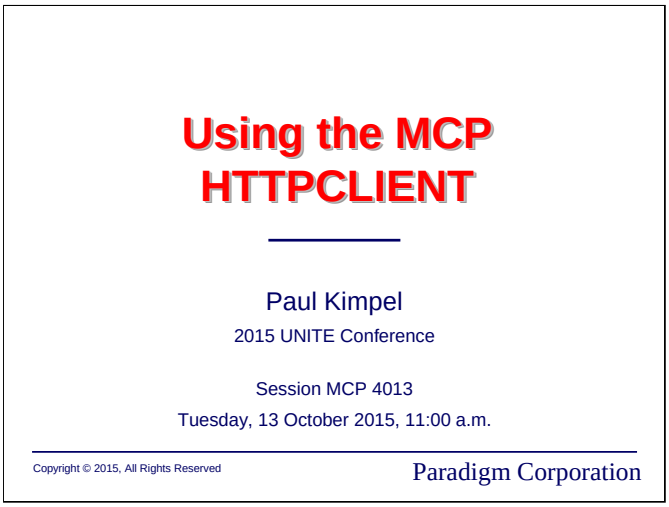


Using the MCP

HTTPCLIENT

2015 UNITE Conference
St. Louis, Missouri

Session MCP 4013

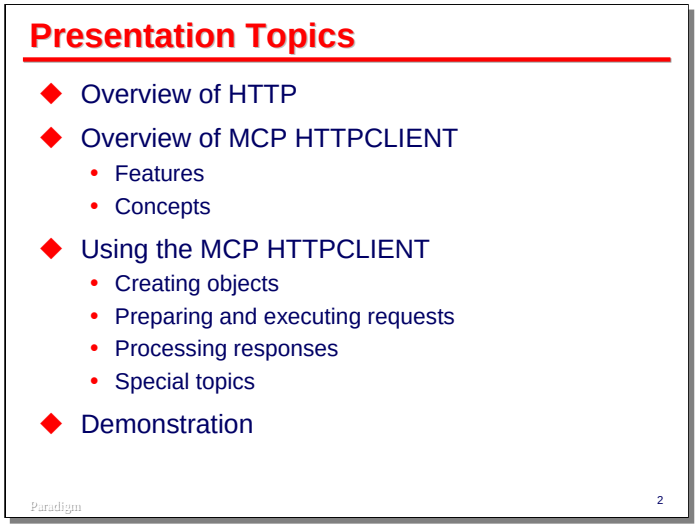
Tuesday, 13 October 2015, 11:00 a.m.

Paul Kimpel

Paradigm Corporation
San Diego, California

http://www.digm.com

e-mail: paul.kimpel@digm.com



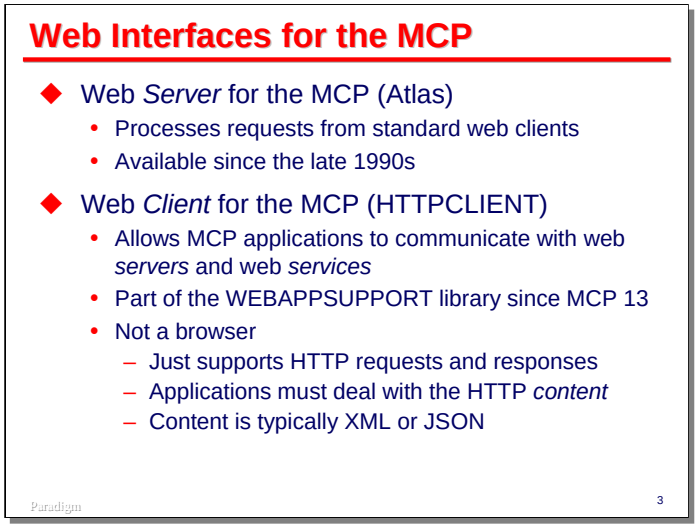
I have been working with a couple of customers on new ways to connect their MCP system to external servers. One of the ways that has proven useful is a relatively recent addition to the MCP, the HTTPCLIENT API. This facility allows an MCP application to send requests to a web server and get a response using HTTP.

I will begin with a brief overview of HTTP, and to make sure everyone understands the terminology, the components of HTTP requests and responses.

Then I will given an overview of the MCP HTTPCLIENT API, focusing on its features and a few of the concepts on which the API is based.

The bulk of the presentation will involve a fairly detailed discussion of the HTTPCLIENT API and how you use it from a COBOL application. This will include creating the objects used by the API, preparing and executing requests to the external web server, processing responses from that server, and a discussion of two special topics, SSL and authentication.

We will wrap up the presentation with a couple of demonstrations of HTTPCLIENT in action.



HTTPCLIENT is a type of web interface, but just to make sure there is no confusion, there are two main types of web interface – client and server.

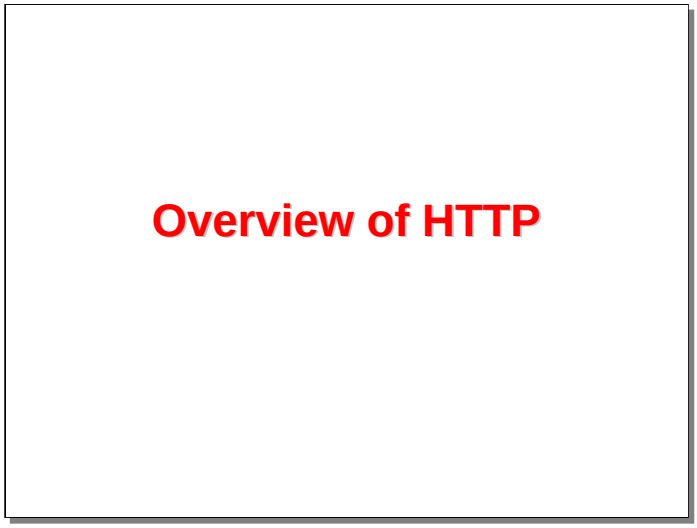
Most people know what a web *server* is – it receives requests from web browsers and sends responses, typically formatted as HTML. Servers can respond with other types of data as well -- images, applets, Javascript code, XML, and JSON. A web server is not limited to these, either. It can send any type of data that can be represented as a stream of bytes.

The MCP has had a web server since the advent of the ClearPath line in the mid-1990s. The current web server product, informally known as Atlas, has been available since the late-1990s.

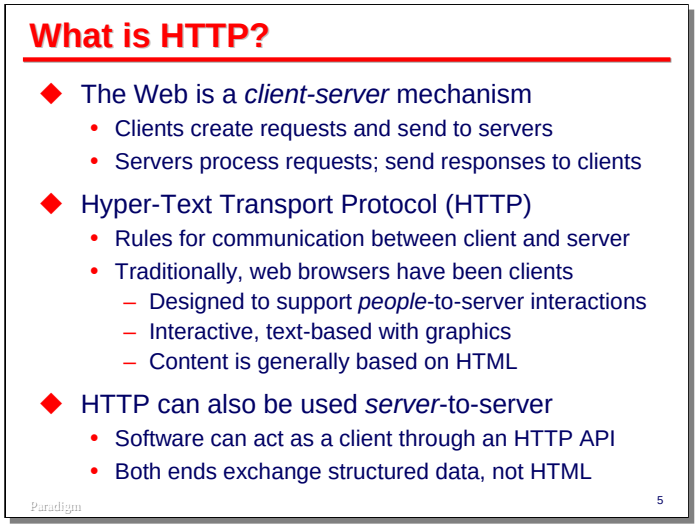
A web *client*, on the other hand, is software that sends requests to a web server and interprets the server's responses. The HTTPCLIENT, as its name implies, is a web client. It allows MCP applications to send requests to web servers and receive the corresponding responses.

HTTPCLIENT is part of the WEBAPPSUPPORT library, which is a component of the Custom Connection Facility (CCF) and was originally developed as a bridge between the Atlas web server and COMS. HTTPCLIENT has been bundled as part of the standard ClearPath IOE since MCP 13.

It is important to note that while HTTPCLIENT operates as a web client, it is not a web browser, at least in the usual sense. It only implements the protocol for communicating with a web server and the facilities to construct requests and process responses. In particular, it does not understand HTML, images, Javascript, or any of the other things we expect a browser to be able to process and activate for our use. Your application can send and receive any of the types of data a normal browser does, but interpretation of that data is the responsibility of your application.



Before we can talk about the HTTPCLIENT, we need to explore some background on HTTP, the **H**yper-**T**ext **T**ransport **P**rotocol.

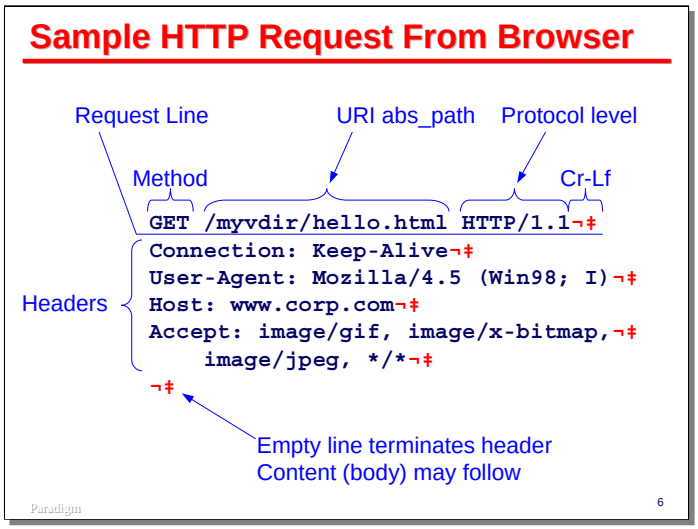


Since we're discussing web clients and web servers, it should be obvious that the web is a *client-server* mechanism. Clients construct requests and send them to servers; servers construct responses and return them to clients. It is essentially a one-in/one-out exchange of data.

HTTP is the protocol by which this exchange takes place. It is a set of specifications for how requests and responses are to be constructed, and how those are exchanged over a network. HTTP operates on top of the TCP/IP protocol.

Originally and traditionally, web browsers have implemented the client role for this protocol. Browsers are designed to support people-to-server interactions. There is normally a person driving the interaction of browser with server, with the responses generally oriented to visual media – text documents, graphics, and video. Responses are generally based on HTML, which may contain references to other types of media. As the browser parses the HTML and encounters these references, it issues requests for the additional media types.

All of that is good, and it's had a tremendous impact on the world over the past 20 years. It did not take long, though, for people to realize that HTTP could just as well for *server*-to-server exchanges, with software application taking the role of a person. Instead of exchanging HTML and visual media, both ends typically exchange data that is structured in a form that an application can easily parse and understand, such as XML or JSON.



This slide shows a very simple HTTP request coming from a browser to a server. The message consists of lines of ASCII text delimited by carriage-return/line-feed (Cr-Lf) character pairs, just as if it had been composed for a teletype printer.

The first line is the "request line." It contains three tokens:

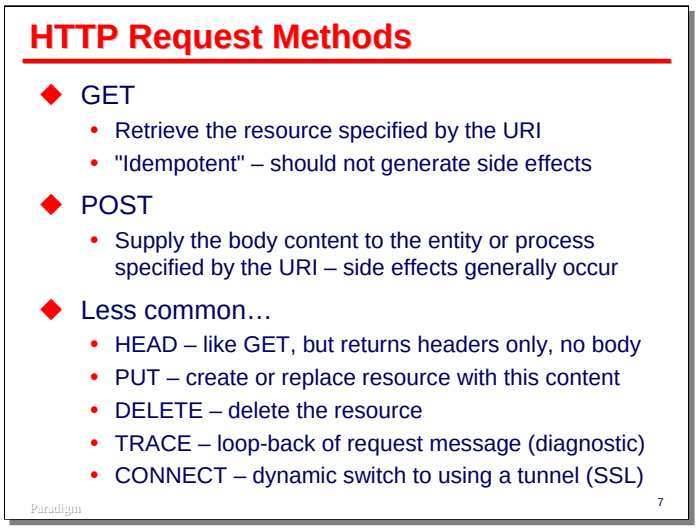
- The Method. This is essentially an HTTP transaction code, and determines the type of message. The most common methods are **`GET`** and **`POST`**.
- The Uniform Resource Locator (URI) absolute path. This identifies the "resource" on the server that the client is trying to access. It may map to a file in the server's file system, or it may identify some other sort of resource, such as a transaction processing program. Although not shown here, the URI path can include a "query string" – name=value pairs that are passed as parameters to the indicated resource.
- The protocol level. HTTP has gone through a number of revisions, and may go through more in the future. Most browsers and servers today use the current HTTP version, 1.1, or its immediate predecessor, 1.0.

Note that the tokens on the request line are simply delimited by a space. Since spaces are sometimes desirable in parts of URI paths, the data must be represented in a special form, called "URL encoding," which we'll discuss shortly.

After the the request line comes an arbitrary number of lines of "headers." These are simply name:value pairs, one pair per line. Headers convey information about the message and instructions for operation of the protocol. You can access these headers, and some of them are quite useful inside application programs. You can look at the HTTP specification to see what the various types of headers are and how they are used.

The header lines are terminated by an empty line—one containing only a carriage-return/line-feed pair.

Some requests may contain "contents" or a body. This will follow the empty line. When a body is present, there will normally be a **`Content-Type`** header indicating the type or format of its data and a **`Content-Length`** header indicating its size.

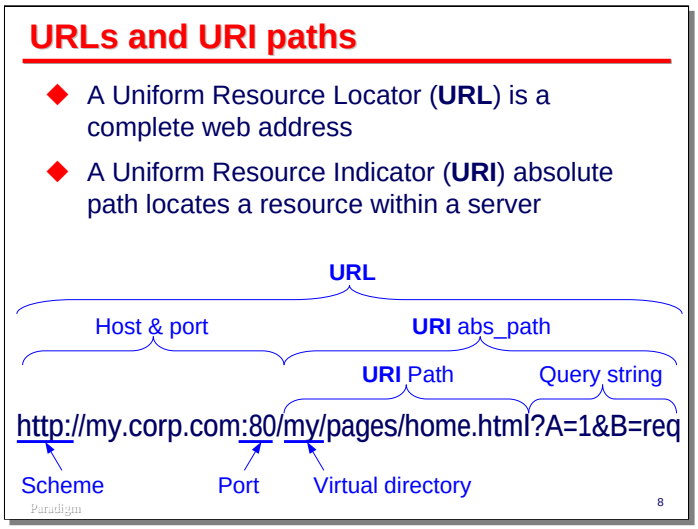


There are several HTTP methods, but two are used in the bulk of requests.

`GET` requests the server to retrieve and return the resource specified by the URI. If the resource is a file, the file is simply sent to the client. If the resource is the name of an executable process, the output from that process is returned to the client. **`GET`** requests are said to be *idempotent* (your word for the day). It means that the result should be the same whether the resource is served from the server itself, or from some intermediate cache – including the cache in the browser. One consequence of this is that resources served by **`GET`** requests should not have any side effects, and should always return exactly the same response, given the same query string and/or content body.

`POST` requests the server to supply the content to the entity or process specified by the URI and to return the unique response to the client. Generally, caching should not occur for responses to **`POST`** requests. **`POST`** is often used with HTML forms to ensure that the request is operated on by the server and not some intermediate cache.

The other methods are less common and not typically used in server-to-server exchanges, unless the server is implementing a RESTful protocol, in which case **`PUT`** and **`DELETE`** are often employed.



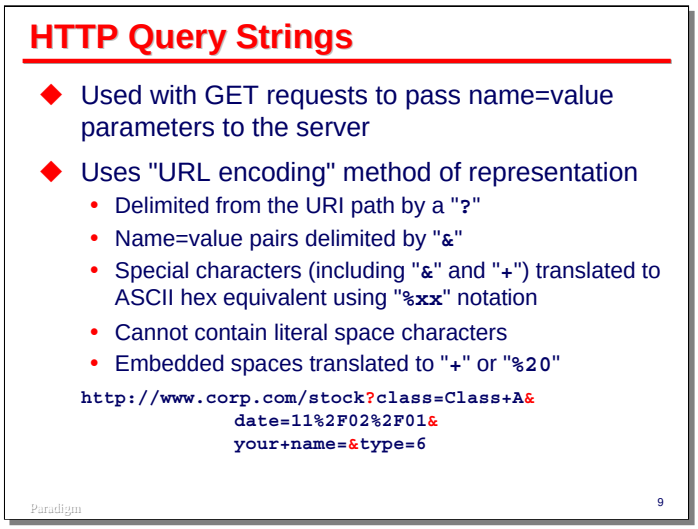
One of the most important concepts in web server environments is that of addressing web resources. The mechanism for this is termed a Uniform Resource Locator (URL), which has as a component the Uniform Resource Identifier (URI) path. These are defined in RFC 2396.

A URL represents a complete web address. It consists of four parts:

- A "scheme" or protocol identifier. For normal web transactions, this is "http." When entering URLs within most browsers, HTTP is the default scheme and its identifier need not be entered. Many web browsers support additional schemes, such as HTTPS, FTP, and gopher.
- The domain name or IP address of the host.
- The TCP port on which the server offers the service. Again, for HTTP, 80 is the default port and does not normally need to be entered. Some web servers offer more than one version of HTTP services, and typically use either alternate port numbers or alternate IP addresses to do so.
- The URI absolute path.

The format of the URI absolute path varies from scheme to scheme. For HTTP, it has the following components:

- The resource path. This is always present, either explicitly or implicitly, and uniquely identifies a resource within the sever environment. The first node of the path is termed the "virtual directory," and typically maps to the base or directory for a collection of resources in the server.
  - For static content, the virtual directory typically maps to a file directory in the server's file system. The rest of the path identifies a specific file subordinate to that directory.
  - For dynamic content, the virtual directory typically identifies some sort of process – perhaps a program, a script, or a transaction code.
- The second part of the URI path is the query string, and is optional. If present, it is delimited from the resource path by a question mark ("**?**") and consists of a set of name=value pairs. Query strings are often used when requesting dynamic content. The name=value pairs are effectively parameters to the process that generates the content. The manner in which the query string is encoded is discussed in more detail on the next slide.

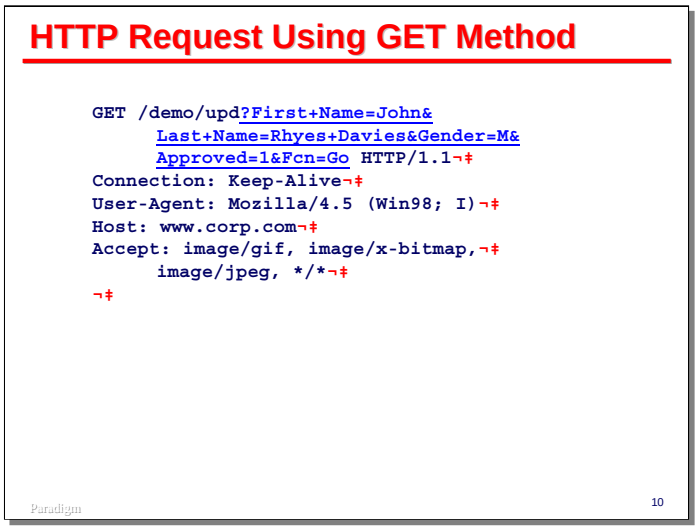


Query strings typically appear in HTTP requests using the **GET** method for a resource that generates dynamic response content. They pass parameters or other control information to the resource. Because the parts of an HTTP request line are delimited by spaces, URLs and URI paths cannot contain literal spaces. Also, some other characters, such as "**/**", "**+**", "**?**", and "**&**" are used in delimiting portions of the URL and URI paths, and cannot appear literally in these entities. In order to allow these reserved characters to be passed in URLs, they are translated using a special convention termed "URL encoding."

Query strings are delimited from the rest of the URI path by a question mark (**?**). They consist of one or more name=value pairs. The pairs are delimited from each other by ampersands (**&**). Within the pair, the name and value are separated by an equal sign (**=**).

Text for the name and value are represented in ASCII. To encode the characters that are reserved, or ones that do not have a visible graphic, the following convention is used:

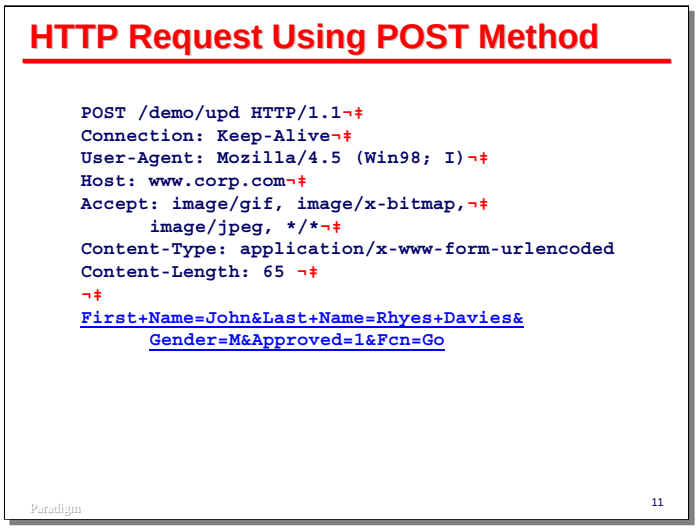
- Space characters are encoded using the plus sign (**+**).
- All other characters are encoded using a percent sign (**%**) followed by two hexadecimal characters representing the character's ASCII code. The space character can optionally be encoded using this method.



This slide shows a **GET** request that contains a query string. Note that the URL-encoded name=value pairs are part of the request line instead of in the body of the message.

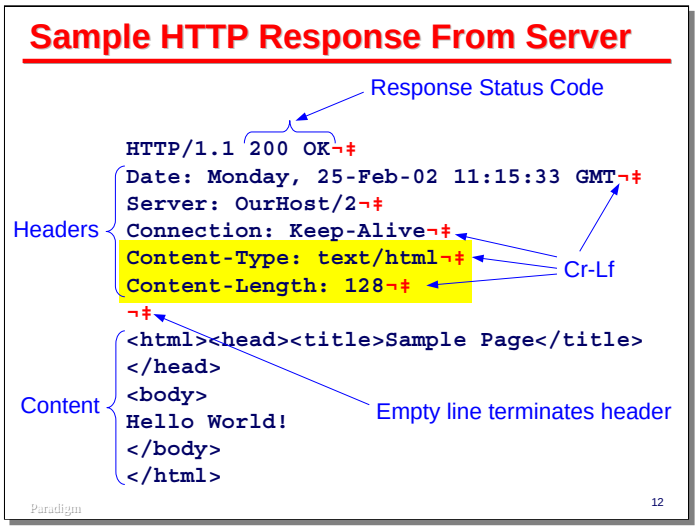
It is up to the designer of the client and the programmer on the server to agree on whether form data should be submitted using **GET** or **POST**. Most clients and servers have an upper limit on how long a request line can be. This is typically at least 1,000 characters, but if you have a form that will send a large amount of data, it may overflow your client or server capacity.

For the MCP HTTPCLIENT, the query string is limited to 2048 characters.



This slide shows a **`POST`** request with a content body. Note the **`POST`** method in the request line, along with the URI path and protocol version.

Also note how the names and values have been translated using URL encoding. Also note the **`Content-Type`** header, which identifies the format of the body, and the **`Content-Length`** header, which informs the server of the content's length.



The format of an HTTP response from the server back to the client is somewhat similar to that of the request.

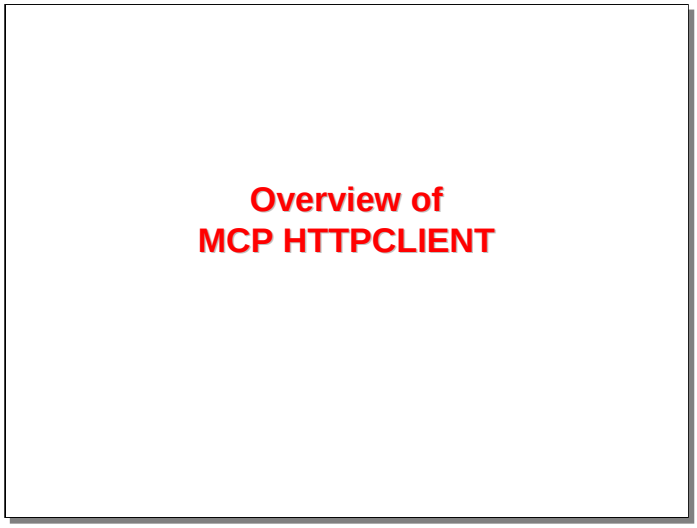
The first line indicates the level of protocol the server is using and a status code. This status code consists of two parts:

- A three-digit number. These codes are defined by the HTTP specification and are intended for easy interpretation by software on the receiving end.
- Descriptive text. The rest of the line after the three-digit number is a text description of the result status, intended for viewing by a person. Most web servers allow you to customize the text for each of the standard response codes. 200 is the default response, and indicates the request was completed successfully. There are quite a few codes defined in the HTTP specification, but the following ones are used most often:
  - 400 = Bad request.
  - 401 = Unauthorized. Sending this will cause the browser to request credentials from the user.
  - 403 = Forbidden operation.
  - 404 = Not found.
  - 500 = Server error processing the request.
  - 501 = Not implemented.

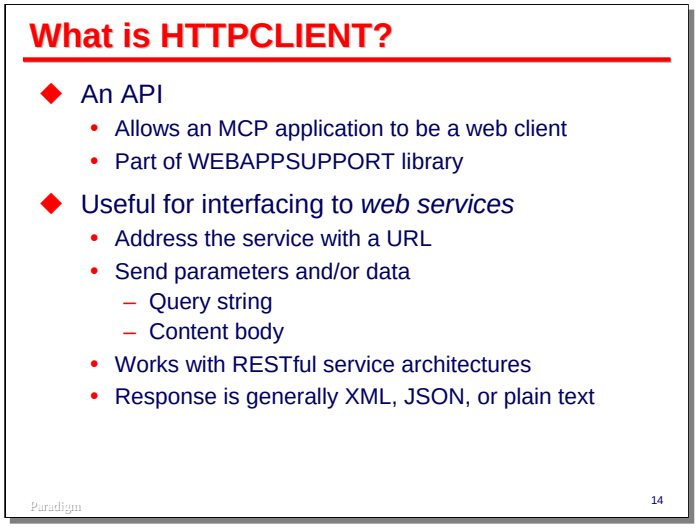
Following the status line are header lines for the response. Many of these are the same for both requests and responses.

As with requests, the header lines in the response are terminated by an empty line.

Following the empty line is the optional content, or body, of the response. Most responses have a body. The slide shows a message body containing HTML, but the body could also be the contents of a text file, a GIF or JPEG image, a Java applet, or some other form of text or binary data. Note the **`Content-Type`** header, which indicates the response is HTML, and the **`Content-Length`** header, which tells the receiver how many octets (bytes) of body to look for.



With that overview of HTTP, let us now examine the features and concepts for the MCP HTTPCLIENT.

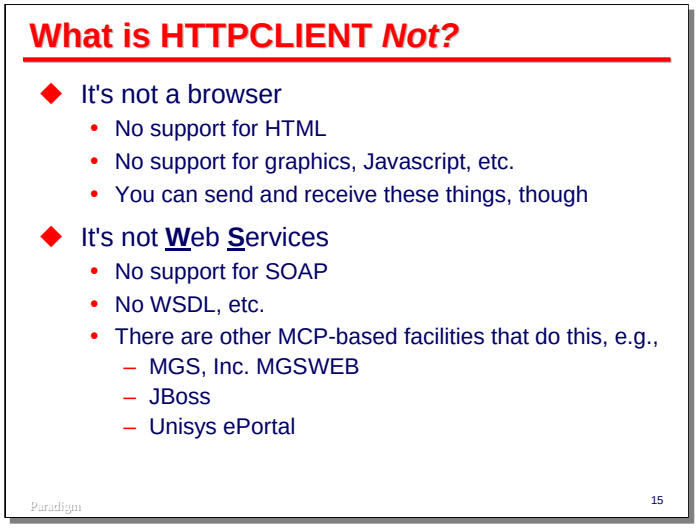


What is the HTTPCLIENT?

First, it is an Application Programming Interface (API) that allows an MCP application to behave as a web client. As previously mentioned, it is part of the WEBAPPSUPPORT library, which in turn is a component of CCF.

Second, it is a useful tool to interface an MCP application to a *web service*. There is no reason you can't do this directly, using TCP/IP port files or the MCP Sockets Service, but HTTPCLIENT hides all of the details of the HTTP protocol and message formats. It provides a more convenient interface for constructing requests, interacting with the web server, and processing responses, especially when using COBOL.

- You address the remote service using a standard URL.
- You can construct requests and send parameters either as a query string or as a content body. HTTPCLIENT provides tools to handle URL-encoding.
- It works quite naturally with RESTful service architectures.
- Web service responses are generally formatted as XML, JSON, or plain text. HTTPCLIENT does not deal with any of these, but it allows you to extract the response data and pass it to some other facility that does, such as the MCP XMLPARSER for XML and JSON.



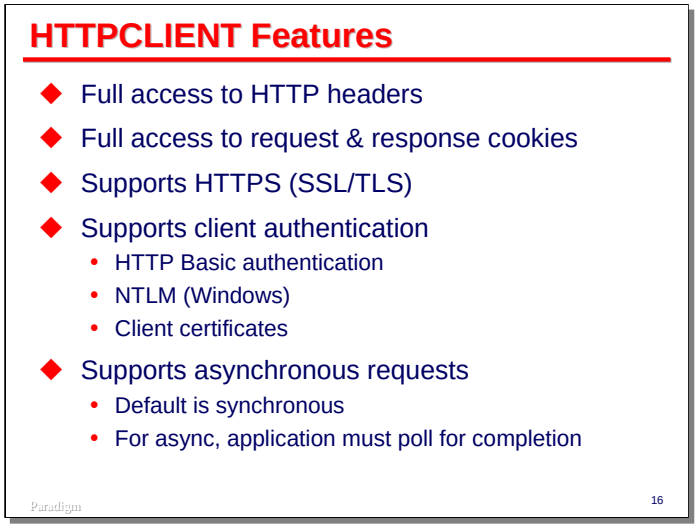
The HTTPCLIENT is definitely not a couple of things as well.

First, while it operates as a web client, it is not a browser. It has no support for processing HTML, images, Javascript, or any of the other media a web browser typically handles. You can exchange this type of data with a web server using HTTPCLIENT, but interpretation and processing of the data is solely your application's responsibility

Second, it is not **W**eb **S**ervices with great, big capital letters. The idea of web services has gone through a number of iterations, most of them producing very heavyweight interfaces. That trend has shifted in recent years to lighter-weight interfaces, for which HTTPCLIENT is more appropriate. In particular, HTTPCLIENT does not support SOAP or WSDL. It is very well-suited, however, for interfacing with RESTful services.

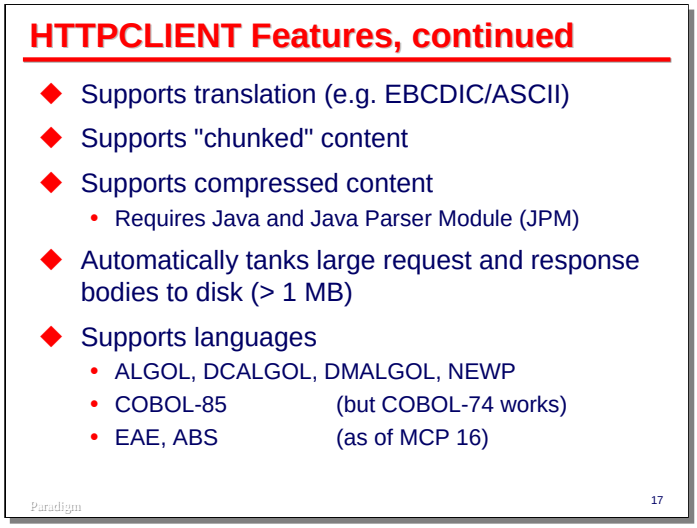
If you need support for heavyweight web services, then there are some solutions available for MCP systems:

- MGS, Inc. has a product called MGSWEB (http://www.mgsinc.com/mcpweb.html).
- The JBoss product distributed by Unisys for MCP systems is all about heavyweight web services.
- Unisys ePortal, being based on Microsoft .Net, has support for heavyweight web services as well.



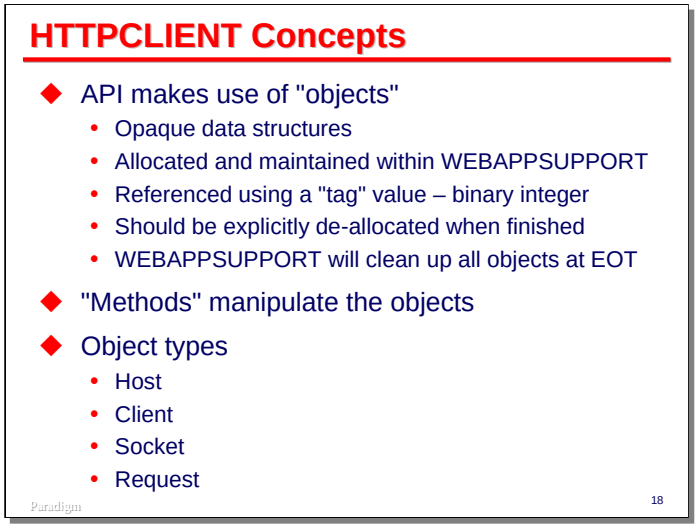
HTTPCLIENT has a number of features that make it useful for communicating with web servers:

- You have full access to HTTP headers in both requests and responses. HTTPCLIENT generates a set of request headers by default. You can modify these, delete them, and insert additional headers in requests.
- You have full access to request and response cookies. You can construct and insert cookies into requests. You can extract and inspect the cookies in a response. By default, HTTPCLIENT will capture the cookies for a response and append them to the cookies for the next request that uses the same client object, preserving any session state the server may have included in the response.
- It supports SSL/TLS. You must have previously installed and enabled SSL in your MCP environment, however.
- It supports three types of client authentication – HTTP "Basic," Microsoft NTLM, and client certificates.
- It supports both synchronous and asynchronous requests. By default, HTTPCLIENT does a synchronous request – your application is suspended while HTTPCLIENT waits for a response from the server. You may enable asynchronous mode, in which case HTTPCLIENT returns to your application immediately after sending the request. Your application must then periodically check the status of the request to determine when it has completed. There is no event mechanism available to signal a status change for asynchronous requests.



Additional HTTPCLIENT features:

- HTTPCLIENT supports translation from and to EBCDIC between your application and the request and response data.
- It supports sending and receiving of "chunked" content. This is typically used when body content is very long or of unknown length. The body is sent in "chunks," with each chunk having a specified length. At the end of the body, a special terminating chunk is sent.
- It supports compression of requests and decompression of responses. In order to take advantage of this, Java and the Java Parser Module (JPM, part of XMLPARSER) must be installed in your MCP environment.
- It automatically tanks requests and responses over 1MB in size to disk. You can construct a large request body with multiple calls from your application, and similarly retrieve a large response body.
- It supports the Algol family of languages, NEWP, and COBOL-85. As of MCP 16, it provides an interface for EAE and ABS. COBOL-74 is not officially supported, but it appears to work.



Using the HTTPCLIENT API involves two entities: objects and methods.

Objects are opaque data structures that hold information about the connection to the web server and about the request and response. These objects are allocated and maintained within WEBAPPSUPPORT, not your application.

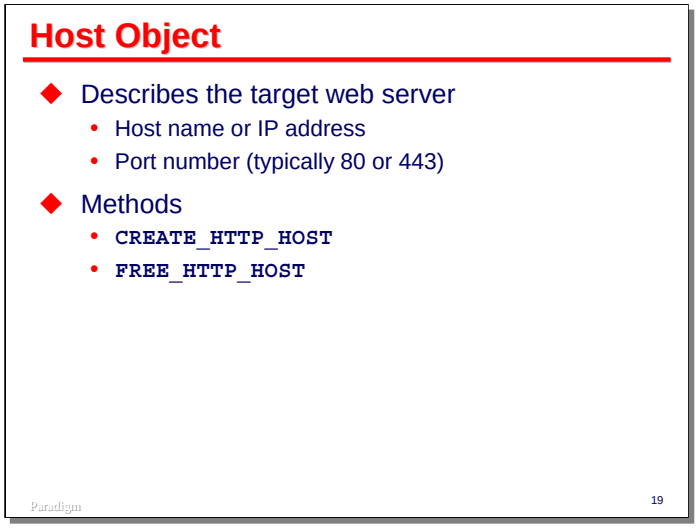
Methods are simply procedure calls into the WEBAPPSUPPORT library. In most cases, these procedure calls operate on one or more objects.

You allocate objects by calling a method in the library. When you allocate an object, a special binary word value, termed a "tag," is returned to you. You then supply this tag value on subsequent method calls to the WEBAPPSUPPORT library to identify the object.

You must explicitly allocate objects before you use them. You must also explicitly deallocate objects when you are finished with them. WEBAPPSUPPORT will, however, implicitly deallocate all objects when your application disconnects from it, typically at end-of-task (EOT). It is not good practice to rely on this, and you should get into the habit of deallocating everything you allocate as soon as you do not need it anymore.

The HTTPCLIENT supports four types of objects. We will discuss each of these in more detail over the new few slides:

- Host
- Client
- Socket
- Request

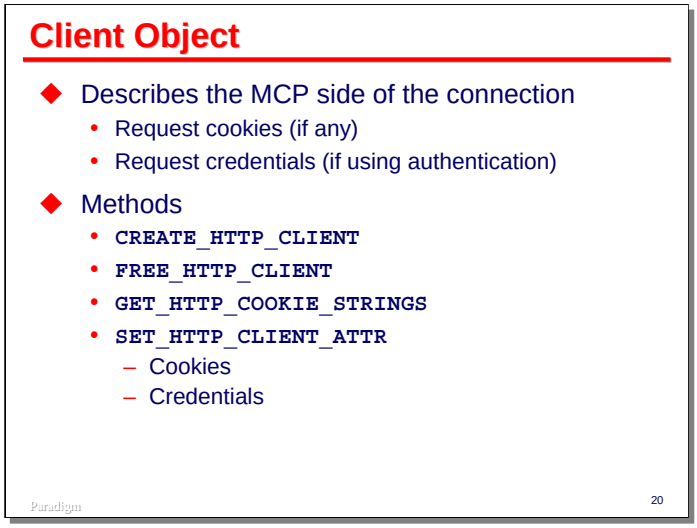


A host object describes the target web server. If your application will be communicating with more than one web server, then either you need to allocate multiple host objects, or allocate them for a specific server as needed and deallocate them when finished.

The host object maintains two attributes of the web server:

- Its host name or IP address
- The port number. For HTTP, the default is port 80; for HTTPS (SSL/TLS) it is normally 443.

The Host object supports only two methods, one to create (allocate) an object (passing the host and port as parameters) and one to free (deallocate) an object.

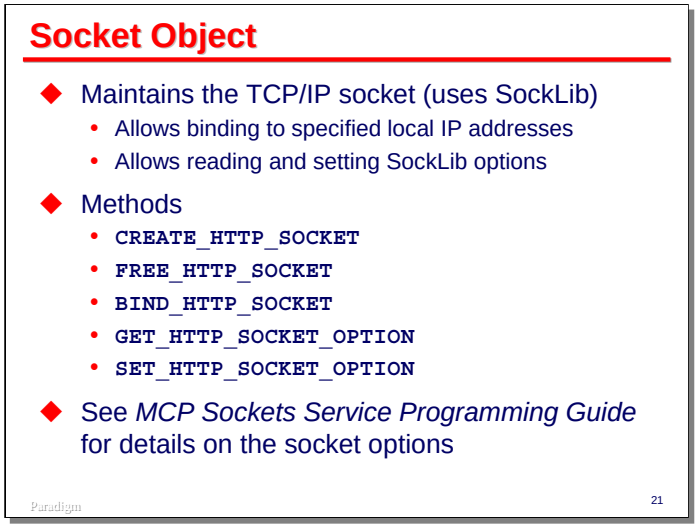


A Client object describes the MCP side of the connection to the web server. Attributes of the client object are optional. There are two types:

- Cookies to be sent with the request
- User credentials to authenticate with the web server

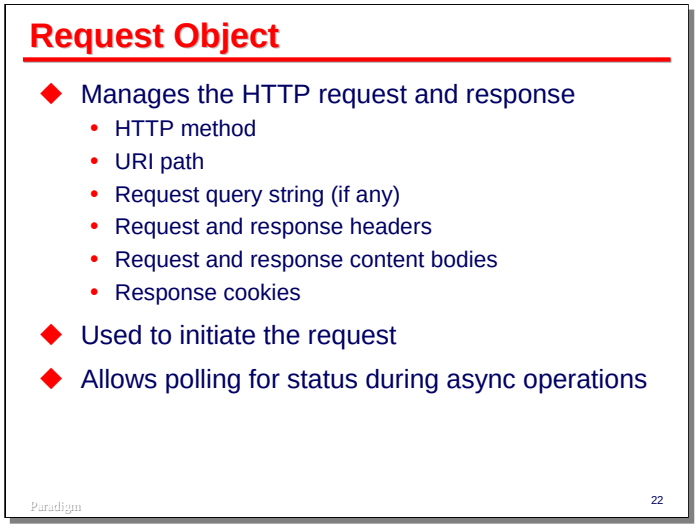
The client object supports four methods:

- Create and free objects.
- Retrieve all cookies currently in the object
- Set cookies and credentials in the object. This can also be used to modify and remove cookies and credentials.



The Socket object maintains the TCP/IP socket used to communicate with the web server. The socket is managed by SockLib (the MCP Sockets Service). Through this object you can get and set socket options, but you do not have access to the socket itself. The Socket object also allows you to bind the connection to a specific local IP address, if it is necessary to ensure that the connection occurs through a certain MCP network adapter.

The Socket object supports methods to create and free objects, bind to a local IP address, and get and set socket options.



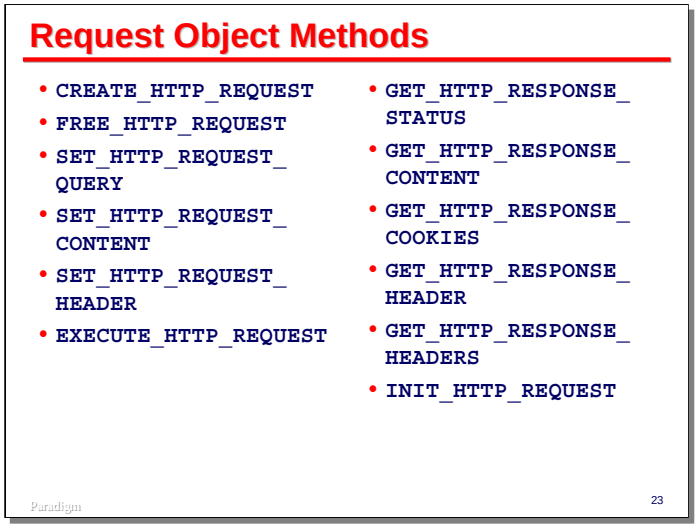
The Request object is the most complex and fully featured of the HTTPCLIENT objects. It manages both the request and response. Through this object you specify:

- The HTTP method (e.g., **`GET`** or **`POST`**)
- The URI path to the server resource
- The request query string, if any
- The request headers, if any
- The request content body, if any

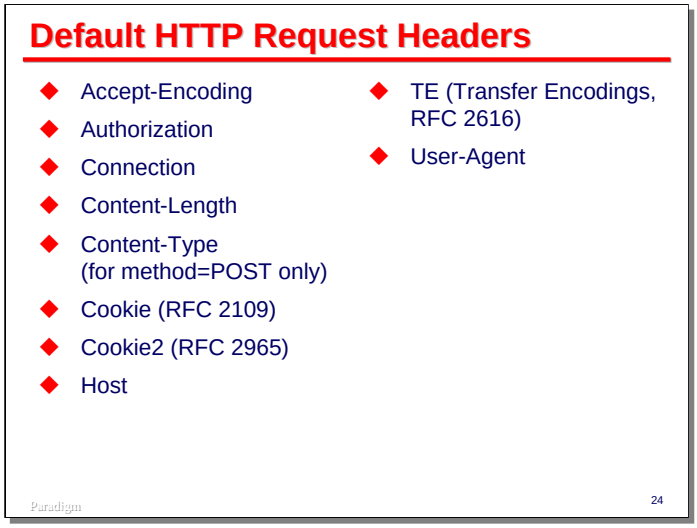
Through this object you also initiate the request and obtain both its completion status and response data. Once the request is complete, this object allows you to retrieve:

- The response content body, if any
- The response headers
- The response cookies

When using asynchronous requests, the Request object allows you to check on the status of the request and determine when it has completed.



The Request object has a large number of methods. We will discuss most of these in detail in the next section of the presentation, but their names give a fair indication of what they do.

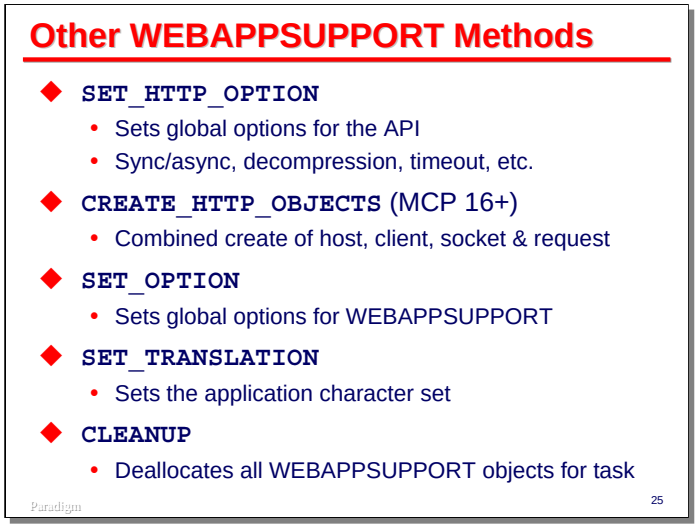


By default, HTTPCLIENT supplies the headers shown on the slide in a Request object. You can modify or delete these default headers, and insert additional headers as necessary, before initiating a request.

In most cases, you do not need to alter these defaults, except possibly for **`Content-Type`**. **`Content-Length`** is normally set for you when you supply a content body for the request. **`Authorization`** and **`Cookies`** are usually established through the Client object, which in turn generates the appropriate headers.

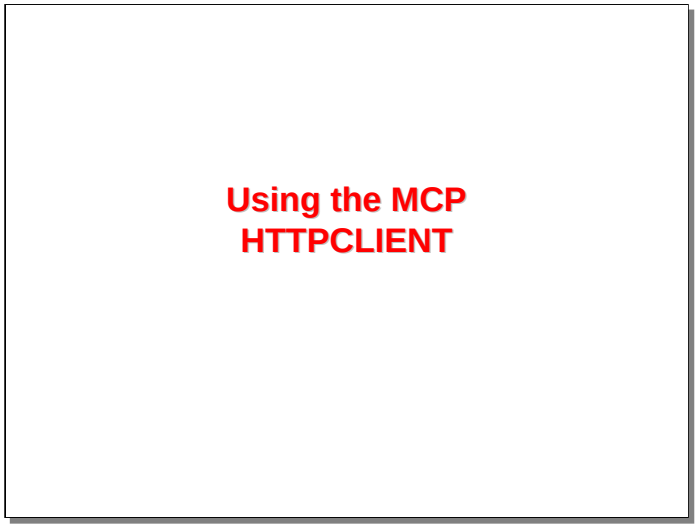
You may wish to modify the Connection header if you need to disable HTTP "keep-alive." One of the examples later in the presentation shows how to do this.

See the WEBAPPSUPPORT reference manual for information on the default value of these headers.

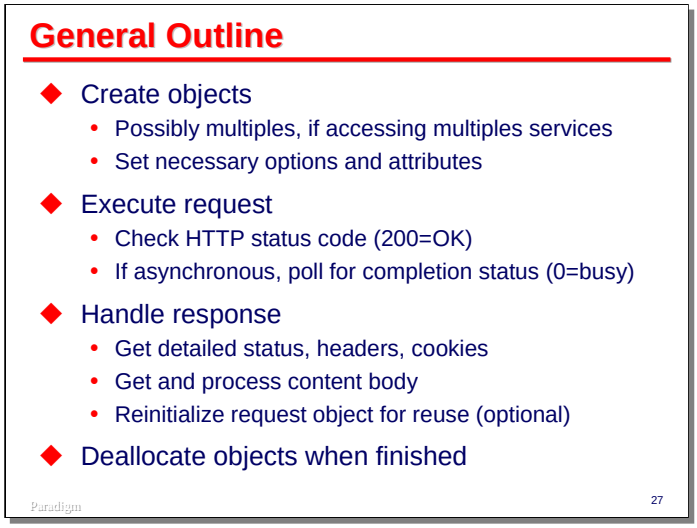


Most of the WEBAPPSUPPORT methods we will be discussing in the rest of the presentation are related to one of the HTTPCLIENT objects. There are several miscellaneous methods, however, of which you should be aware:

- **SET_HTTP_OPTION** sets a number of global options for the API. Using this method you can specify whether to do synchronous or asynchronous requests, whether the API will automatically compress and decompress data, the amount of time the API should wait for a server response before generating a timeout error, etc. See the documentation for details on all of the options.
- **CREATE_HTTP_OBJECTS** is a convenience method available starting in MCP 16. You always need to allocate at least one of each of the Host, Client, Socket, and Request objects in order to initiate a request. Each object has its own create method, but this method will create all four in one call.
- **SET_OPTION** sets global options for the WEBAPPSUPPORT library. It is used to specify file options and attributes, compression options, and cache controls.
- **SET_TRANSLATION** is used to specify the application character set. By default this is EBCDIC, but can be changed to ASCII or UTF-8.
- **CLEANUP** will deallocate all WEBAPPSUPPORT objects associated with your task. It is a good idea to call this during your application wrap-up code.

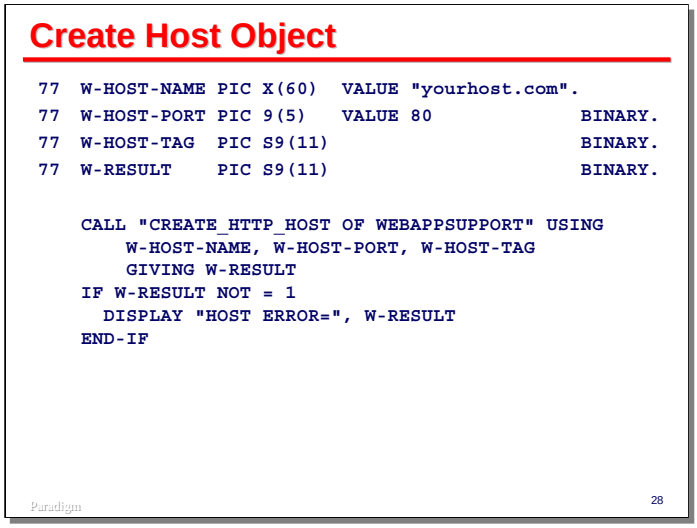


With the foregoing as background, we will now explore how you use the HTTPCLIENT to interact with a web server.



This slide shows a general outline of the steps to prepare a request and obtain a response using HTTPCLIENT:

- First, you need to create the necessary objects. You will need at least one each of the Host, Client, Socket and Request objects. If your application must communicate with multiple web servers, or access multiple URIs, you may need to create more than one instance of some object types.
- Next you need to set any necessary attributes for the objects, including headers, cookies, authorization credentials, socket options, query string, and content body.
- One everything is set up, you execute (initiate) the request. If using synchronous requests (the default), you should always check the HTTP status code that is returned. A value of 200 indicates a successful request. The 20x-series of status codes indicate warnings. If you are using asynchronous requests, your application will need to poll periodically for completion status. A value of zero indicates the request is still in progress.
- Once the request is complete you need to handle the response. This may entail any of:
  - Getting detailed response status
  - Retrieving the values of headers and cookies
  - Retrieving and processing the content body
- If you plan to reuse the Request object for another request, you must reinitializing the Request object. This leaves all headers and cookies intact. You will typically need to supply a new query string or content body, however.
- Finally, when you are finished with an object, you *must* explicitly deallocate it.

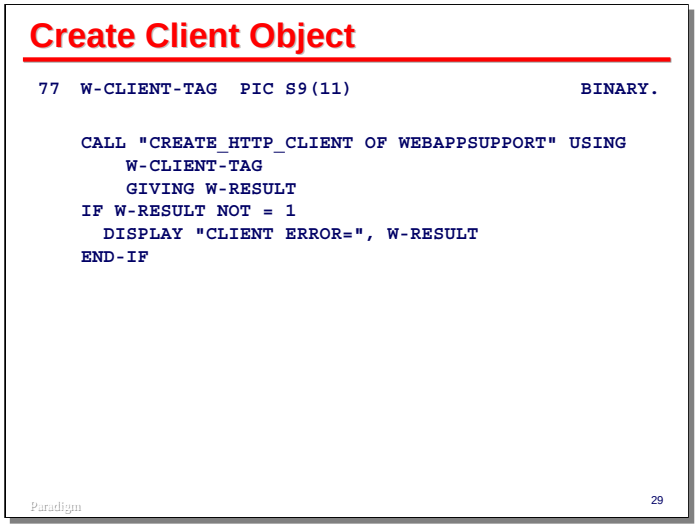


The first object we will create is the Host object. This requires the name of the host (as either a DNS name or an IP address) and the port on which the server will be listening. For HTTP, the default port is 80; for HTTPS (SSL/TLS), the default port is 443.

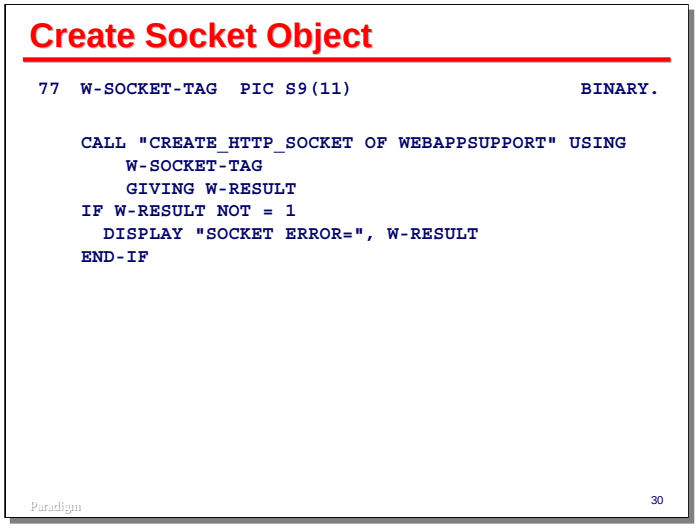
The slide shows how you call the create method for the Host object, passing the host name, port, and a variable that will return the "tag" value for the object.

Most WEBAPPSUPPORT methods return a binary integer value that indicates the success or failure of the method call. Value 1 indicates success; negative values indicate an error, as documented in the WEBAPPSUPPORT manual; for some methods, the value zero indicates no operation was performed.

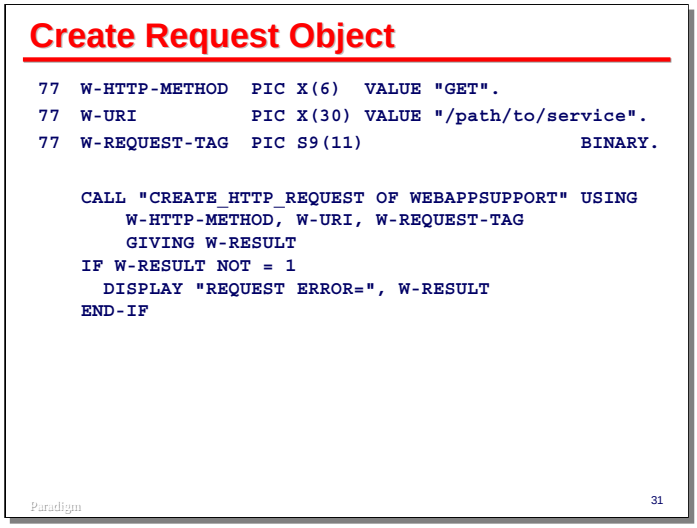
You need to save the host tag value for use in later method calls.



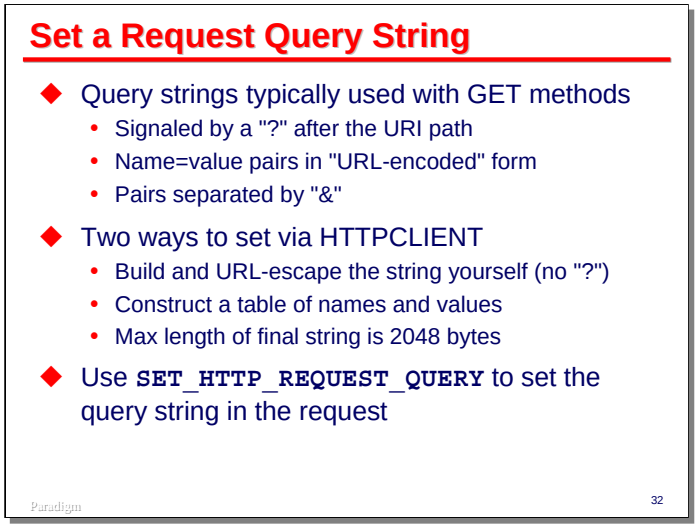
The second object we will create is a Client object. This is a very simple call, requiring only a variable to return the object's tag value. Cookies and authentication credentials are set in this object through another method call, discussed later.



The third object we will create is the Socket object. This is also a simple call, requiring only a variable to return the object tag.



The fourth, and final object to create is the Request object. This method requires parameters to specify the HTTP method name and the URI path to the resource on the web server. Note that you do not specify the scheme, host name, port, or query string here, just the URI path. The create method returns the tag of the newly-created object.

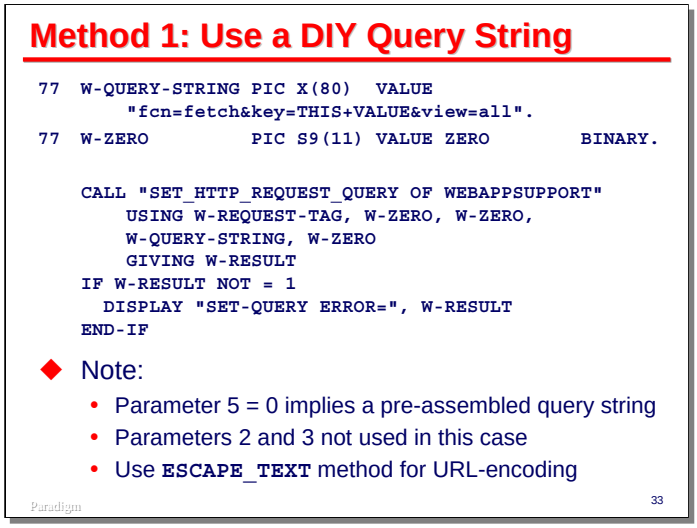


One of the most common things you need to do in preparing a request, especially with **GET** requests, is specify a query string. In the request line of the outgoing HTTP request, the presence of a query string is signaled by a "**?**" immediately following the URI path, followed by the query string's name=value pairs in URL-encoded form.

There are two ways to set a query string using the HTTPCLIENT:

- Build and URL-escape the query string yourself. You do not include a leading "**?**" in this string -- the API will do that for you.
- Build a table of name and value strings. In COBOL, you simply define a record area with an **OCCURS** group item. Subordinate to that group item, you define a name item of a certain length and a value item of a certain length. When calling the **SET_HTTP_REQUEST_QUERY** method, you specify the number of pairs and the width in characters of the name and value "columns" in the table. The method will URL-escape the names and values for you, and concatenate them into a query string.

Once you have either the query string or the table of names and values built, you call **SET_HTTP_REQUEST_QUERY** to insert the query string into the Request object.

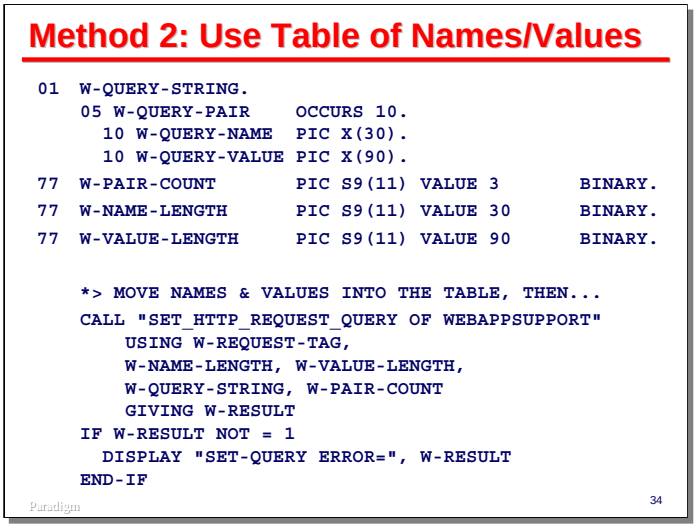


This slide shows an example of the Do-It-Yourself query string approach.

- The method takes the Request object tag as a parameter.
- The next two parameters are used with the table approach and are ignored in this case, so we simply pass zeroes.
- The fourth parameter contains the URL-escaped text of the query string, again without the leading "**?**" character.
- The fifth parameter indicates the number of name/value pairs when using the table approach. If the value of this parameter is zero, it indicates you have built and URL-escaped the query string yourself.

Note that you must URL-escape the names and values individually. If you build the entire query string and then escape it, the "**=**" and **"&"** delimiters will also be escaped, and the web server will not see what you intended.

WEBAPPSUPPORT has a utility method that will URL-encode the text in a data item – **ESCAPE_TEXT**. See the WEBAPPSUPPORT documentation for instructions on its requirements and parameters. There is also an example of its use in the **HTTPCLIENT/GEOCODER/DEMO** sample program accompanying this presentation.

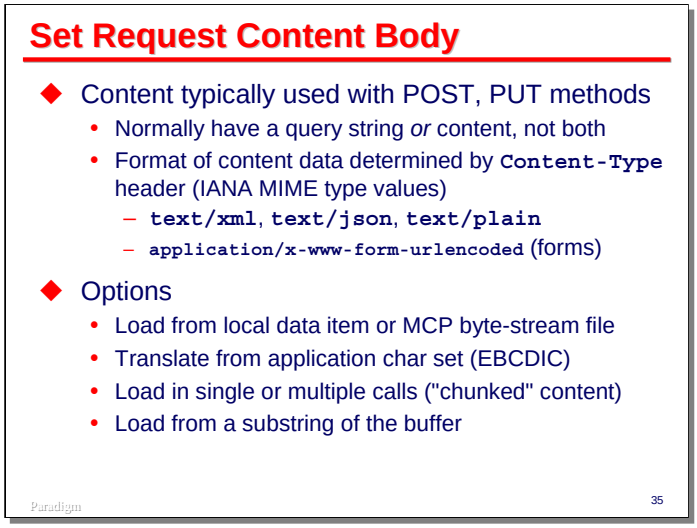


This slide shows an example of the table method to construct a query string.

Note the structure of the **W-QUERY-STRING** record area that will be used as a table of names and values. Also note that the values of **W-NAME-LENGTH** and **W-VALUE-LENGTH** match the size of the name and value columns in this table, respectively.

The **SET_HTTP_REQUEST_QUERY** method call is exactly the same as for the prior approach, except that the fifth (**W-PAIR-COUNT**) parameter specifies the number of entries in the table and the second and third (**W-NAME-LENGTH**, **W-VALUE-LENGTH**) parameters specify the width of the name and value columns in the table.

Note that in this case, you do *not* URL-escape the name and value fields; the method will do that for you.

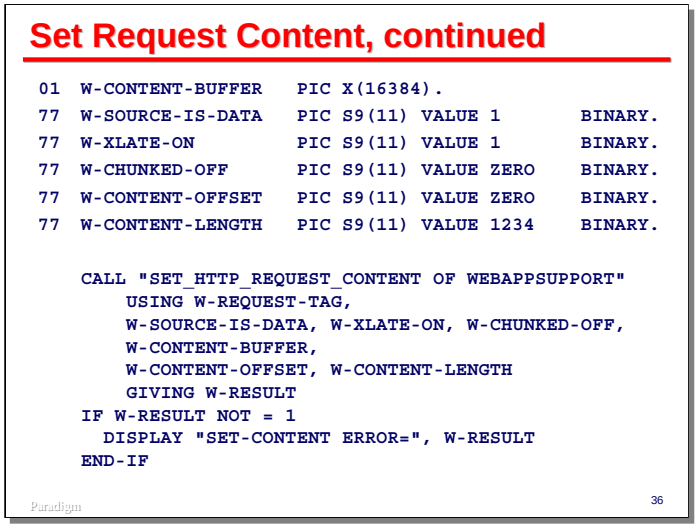


Although a request can have both a query string and a content body, that is not common. Typically requests have one or the other. Content bodies are typically used with **`POST`** and **`PUT`** methods.

The format of the body is determined by the **`Content-Type`** header. This is a MIME-type string, as defined by the IANA. The slide shows some common examples of MIME types. Note that when you specify **`POST`** as the method for a Request object, the API will by default supply a **`Content-Type`** header with the value **`application/x-www-form-urlencoded`**. This is the value used by web browsers for data from HTLM forms. You must override this default if your body is of a different type.

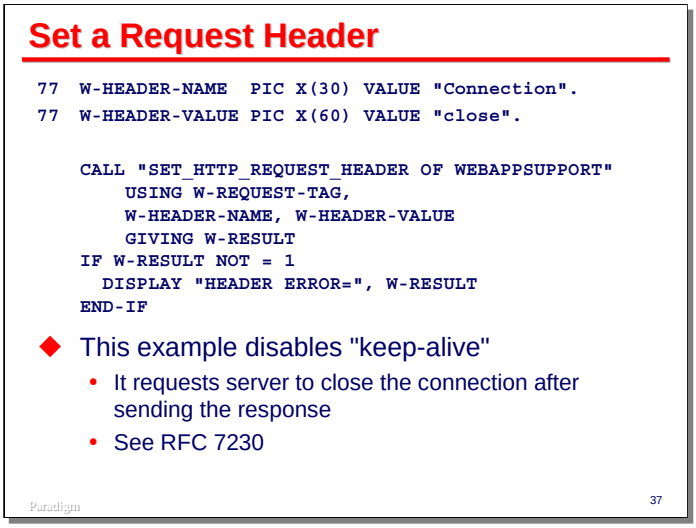
You can specify several options when setting the content body for a request:

- You can specify whether the body data will come from a data item in your application, or from an MCP file. The file must be a byte stream, not a record-oriented file.
- You can specify whether the body data should be translated from EBCDIC to ASCII.
- You can specify whether the body data will be supplied all at once, or through multiple calls to the **`SET_HTTP_REQUEST_CONTENT`** method. Supplying the data through multiple calls generated a "chunked" request.
- You specify where in the data item the body data or MCP file name occurs – its zero-relative offset and length in bytes.



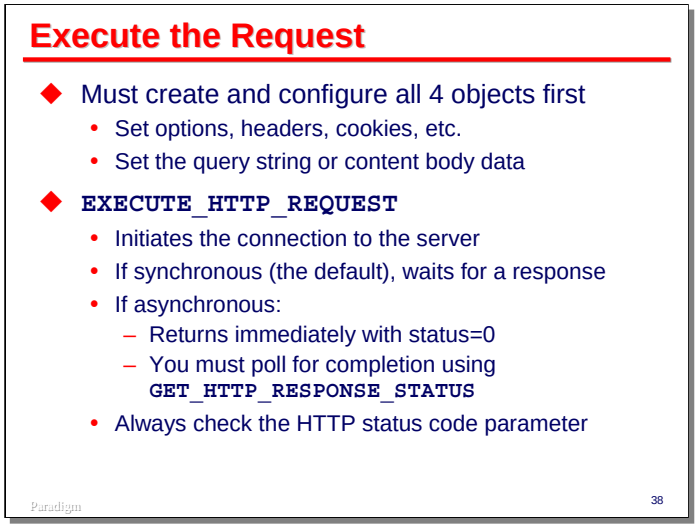
This slide shows an example of setting content body data:

- The first parameter is the Request object tag.
- The second parameter indicates the body data will come from an application data item, not an MCP file.
- The third parameter indicates the data will be translated from EBCDIC to ASCII.
- The fourth parameter indicates the body will be supplied in one call (i.e., non-chunked).
- The fifth parameter supplies the body data. If the data were to come from an MCP file, this parameter would supply the title of that file
- The fifth and sixth parameters specify the data starts at the beginning of **`W-CONTENT-BUFFER`** (offset zero) and has a length of 1234 bytes.



This slide shows an example of setting a header value in the Request object. You specify the object tag, header name, and header value as parameters. The API will trim trailing spaces from the name and value parameter strings.

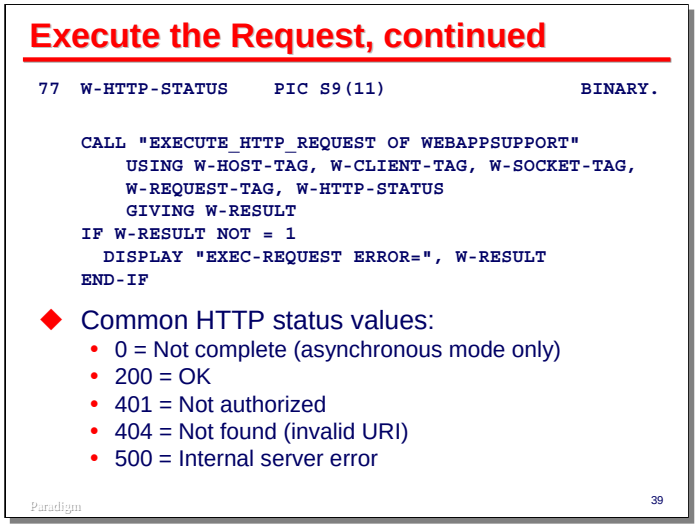
This example also illustrates how to turn off the "keep-alive" mechanism. Connections to the web server are normally closed by the server. In HTTP 1.1, the server will by default keep the connection open, anticipating additional requests from the client (e.g., to retrieve images or scripts). By setting the **`Connection`** header to the value **`closed`**, the client requests the server to close the connection once the response has been sent.



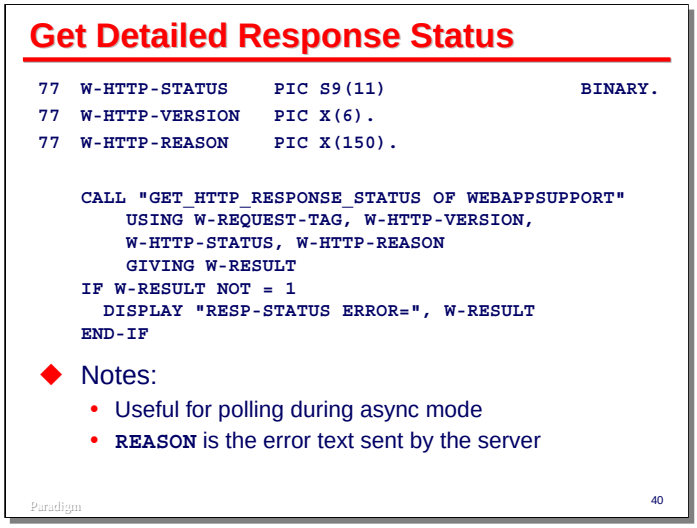
Once all of the objects have been set up and their attributes established, you are ready to execute the request. By default, requests are executed in synchronous mode. One of the options of the **SET_HTTP_OPTION** method will change this to asynchronous.

The execute method opens the connection to the server. If synchronous, it will wait for a response from the server, or until the timeout period expires. If asynchronous, the method will return immediately to the caller with a status value of zero, and the caller must then periodically check the status of the request using **GET_HTTP_RESPONSE_STATUS** to determine when it has completed.

Regardless of the mode in which the request is executed, you should *always* check the response status to determine whether the request completed successfully. 200 indicates a successful completion.

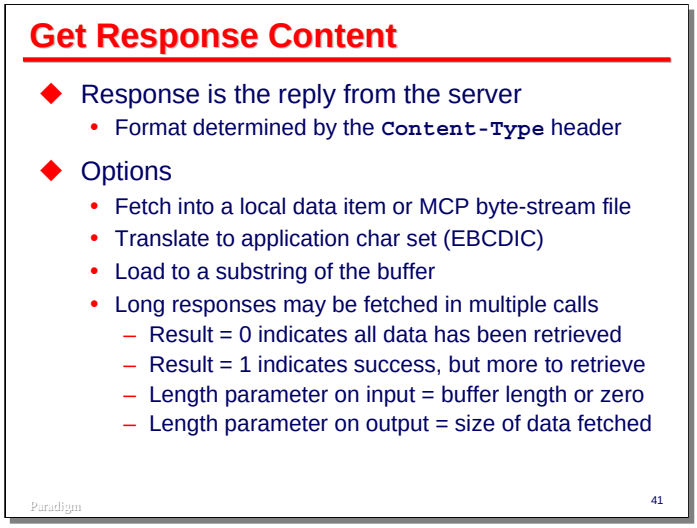


This slide shows an example of a call on the execute method. You simply pass the tag values for the Host, Client, Socket, and Request objects. The method will return with a value in the status parameter.



The status code returned by **`EXECUTE_HTTP_REQUEST`** is just a binary integer. You can get more detailed status information by calling the **`GET_HTTP_RESPONSE_STATUS`** method. You would normally call this after a request has completed, but when executing an asynchronous request, you also use this method to poll for completion status.

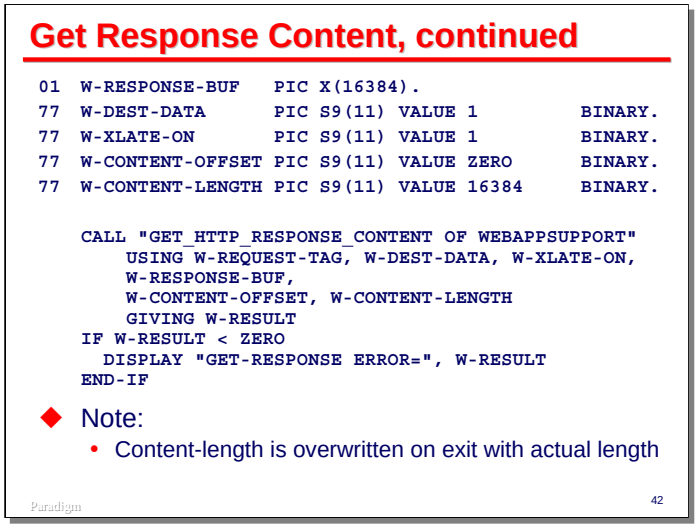
You call this method using the tag value of the Request object. It returns the same status code as the execute method. It also returns two string-valued results, the version of HTTP used by the server, and any text supplied by the server as a reason for the status code. The reason for a successful (200) status is usually just "**`OK`**".



Most HTTP responses have a content body, so your application will typically want to retrieve that data and process it. Your application may also wish to retrieve at least the **`Content-Type`** header, which describes the format or representation of the data, e.g., **`text/xml`**, text/**`plain`**, **`text/json`**, etc.

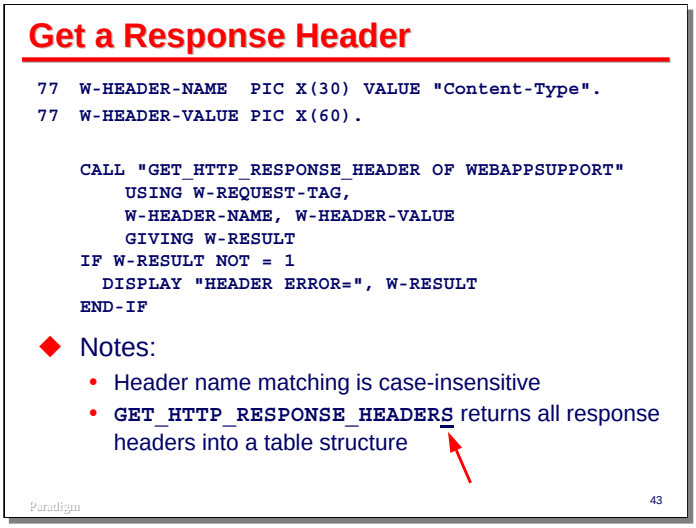
There are several options you can specify that control how the content body data is returned to your application:

- You can fetch the data into a local data item, or have it stored in an MCP file. The file will be a byte stream.
- You can specify that the data be translated from ASCII (or however the server returned it, e.g., UTF-8) to EBCDIC.
- You can specify where in the data item the data should be fetched by means of a zero-relative offset.
- If the response is large, you can fetch the data using multiple calls. In this case, a result code of 1 from the method means that the current call was successful, but there may be more data. A result code of zero indicates that all data has been retrieved.
- Note that in all cases, the length parameter on input defines the number of bytes available in the data area where body data can be stored. On output, the length parameter indicates the number of bytes actually stored.



This slide shows an example of retrieving body content data into an application data item.

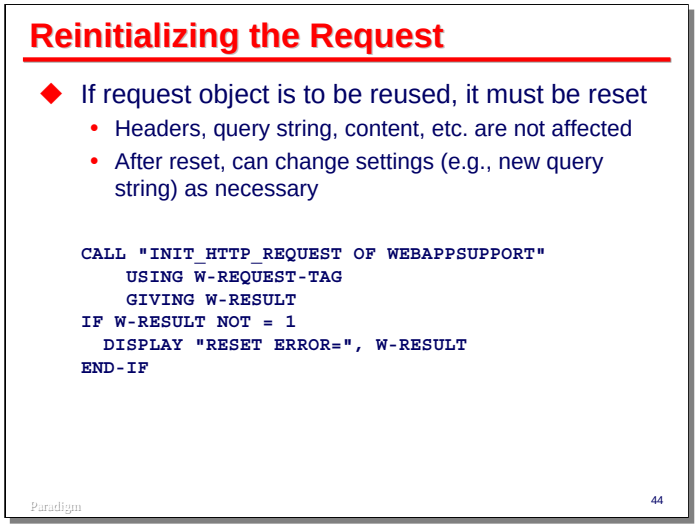
- The first parameter is the tag for the Request object.
- The second parameter indicates the data will be fetched into the **`W-RESPONSE-BUF`** parameter rather than stored in an MCP file.
- The third parameter specifies the data will be translated to EBCDIC.
- The fourth parameter is the item where the body data will be stored. If the data is being retrieved into an MCP file, the your application will store the title of the file in this item instead.
- The fifth parameter indicates the zero-relative offset into the fourth parameter where the data will start.
- The six parameter on input indicates the maximum number of bytes available in the fourth parameter. On output (if the data is being stored into the fourth parameter) it will indicate the number of bytes actually stored.



This slide shows an example of retrieving a header value from the response:

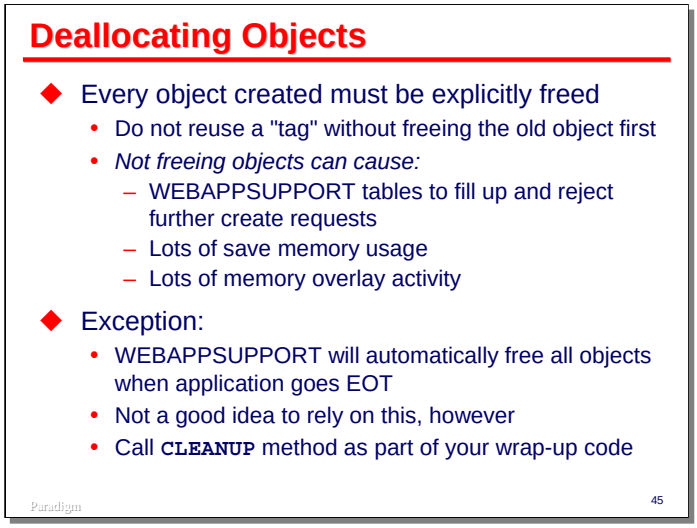
- The first parameter is the tag for the Request object.
- The second parameter is a space-terminated string containing the header name. Header names are case-*insensitive*.
- The third parameter is a data item to receive the header value. This item will be right-filled with spaces.
- Both header name and value will be translated to the *application* character set – EBCDIC by default.

In addition to **`GET_HTTP_RESPONSE_HEADER`**, which retrieves a specified header, there is the **`GET_HTTP_RESPONSE_HEADERS`** method. This retrieves *all* headers from the response into a COBOL-like table of names and values, similar to the table used to build query strings. See the WEBAPPSUPPORT documentation for details.



After completing a request, you can reuse the Request object. Before doing so, however, you must first reset it using the **`INIT_HTTP_REQUEST`** method. This resets the internal state of the object, but attributes of the object, such as headers and the query string, remain as they were.

Resetting a Request object is much more efficient than deallocating and then reallocating one.



Because the WEBAPPSUPPORT objects must be explicitly allocated, they should be explicitly deallocated when you are finished with them. This will free up both memory and table slots in WEBAPPSUPPORT.

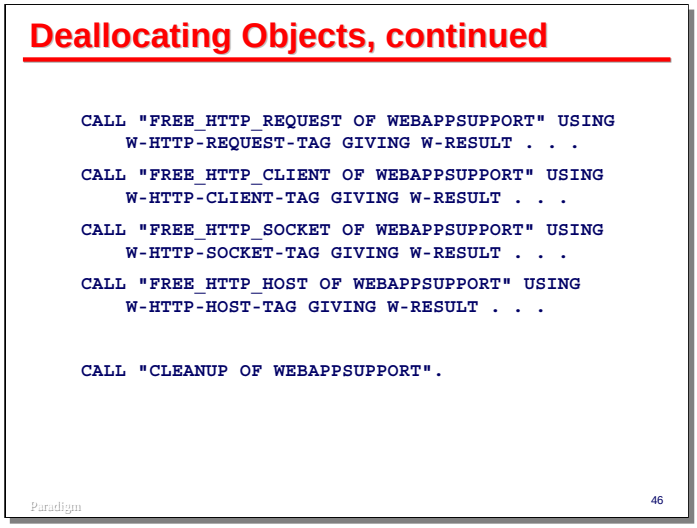
One serious error in programming an HTTPCLIENT application is to allocate an object, and then allocate a new object using the same tag variable without deallocating the original object first. When you do that, the original object continues to exist, but your application loses the tag value that allows you to reference that original object.

If this is done repeatedly, it can result in excessive memory usage (especially save-memory usage) by the WEBAPPSUPPORT library. Eventually the library may run out of table slots with which to manage the objects (each type is limited to 64K instances), at which point it will refuse to allocate any more until some are deallocated.

WEBAPPSUPPORT will deallocate all objects allocated by your program when it disconnects from the library. This typically occurs at program EOT. If you have a long-running program that is allocating new objects without deallocating its old ones, it could exhaust all of the available table slots.

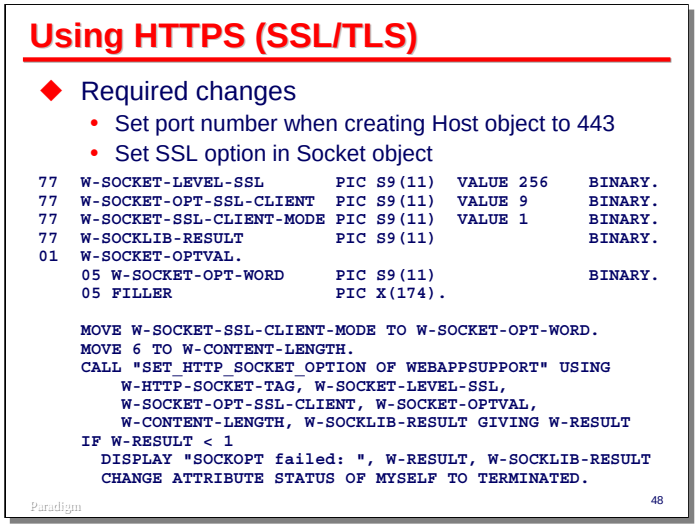
Therefore, you need to keep track of the objects you have allocated and explicitly deallocate them when they are no longer needed.

You can perform the same type of mass deallocation that happens when you disconnect from the WEBAPPSUPPORT library by calling its **`CLEANUP`** method, which takes no parameters.



This slide shows examples of all of the object deallocation method calls, as well as the **`CLEANUP`** method.

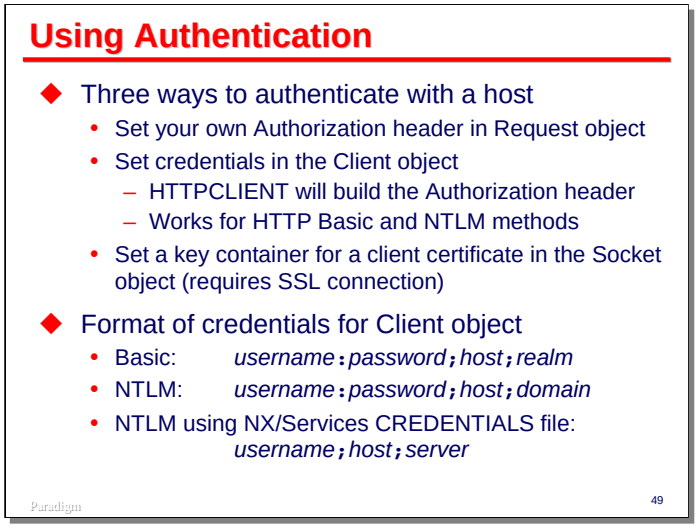
Before wrapping up this discussion on the HTTPCLIENT, there are two special topics to consider – using SSL/TLS and supplying authentication credentials.



There are two small changes you need to make when using the HTTPCLIENT over HTTPS:

- When creating the Host object, you need to specify the appropriate port number. The default port for HTTPS is 443.
- After creating the Socket object, you must set the SSL option in that object. The slide shows sample code to accomplish this.

Of course, in order for SSL to work, the MCP Cryptography Services (CryptoProxy) software must first have been installed and configured properly, including importing the necessary root trusted Certificate Authority certificates. You must also enable SSL in TCP/IP.

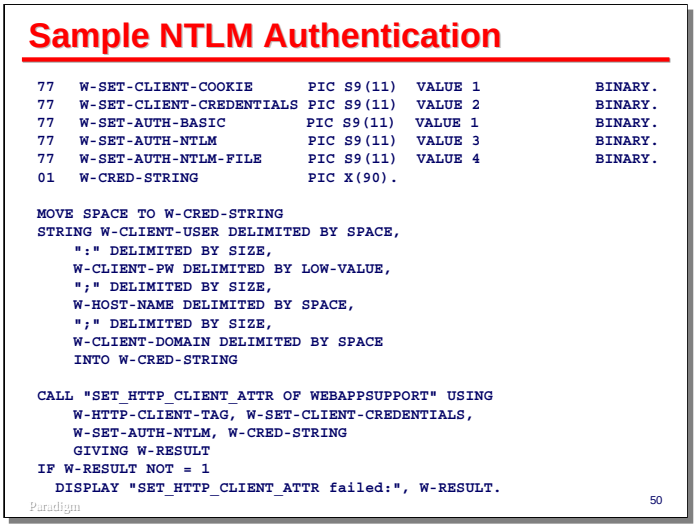


The HTTPCLIENT supports three ways for you to authenticate with a web server:

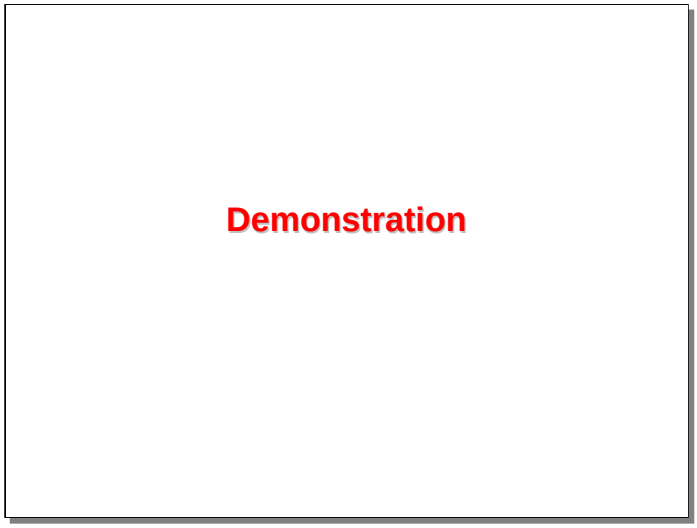
- Set your own Authorization header in the Request object.
- Set client credentials in the Client object. Using this method, the HTTPCLIENT supports HTTP "Basic" and NTLM authentication methods, and will build the appropriate Authorization header itself.
- If you are using HTTPS, you can set a key container for a client certificate in the Socket object.

If using HTTP Basic or NTLM authentication, the credentials are supplied to the API in the form of a string. The slide shows how that string should be formatted for each authentication method.

Note that there are two ways to supply credentials for NTLM authentication. The first is to include the password in the credentials string. The second, more secure, method is to reference an NX/Services **`CREDENTIALS`** file that contains an encrypted password. In this case, "host" is the host name of the web server; "server" is the value identifying the server-name node in the **`CREDENTIALS`** file name.



This slide shows an example for configuring NTLM authentication by specifying the password as part of the credentials string.



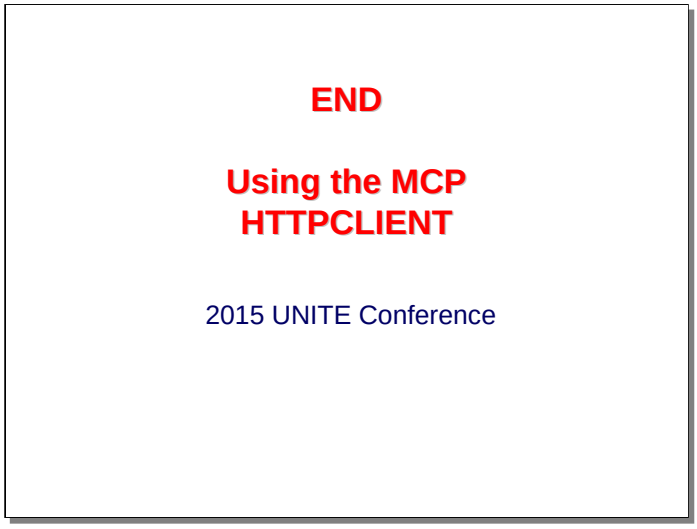
I found two free web services sites that do interesting things, and wrote a couple of COBOL-85 programs to interact with those sites. Those programs are in the associated files for this presentation. You can find out more about these sites and the services they offer at their respective web sites:

- http://www.geonames.org/
- http://www.geocoder.us/

References

- *WEBAPPSUPPORT Application Programming Guide* (3826 5286), Section 9
- HTTP 1.1 specifications
  - https://tools.ietf.org/html/rfc7230
  - https://tools.ietf.org/html/rfc7231
  - https://tools.ietf.org/html/rfc7232
  - https://tools.ietf.org/html/rfc7233
  - https://tools.ietf.org/html/rfc7234
  - https://tools.ietf.org/html/rfc7235
  - https://tools.ietf.org/html/rfc7236
  - https://tools.ietf.org/html/rfc7237

Paradigm 52


END
Using the MCP
HTTPCLIENT
2015 UNITE Conference