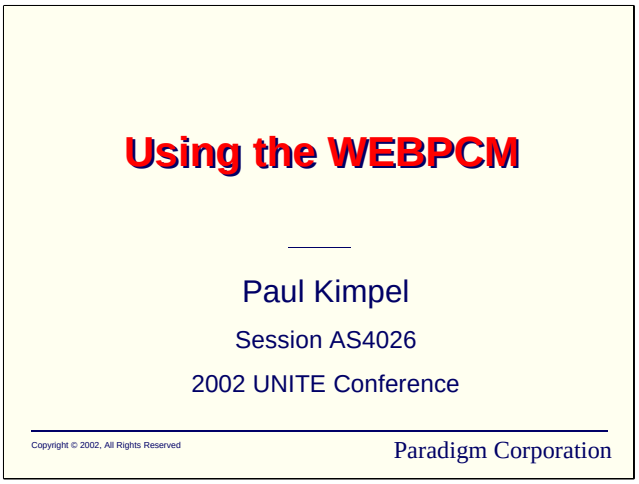


Using the WEBPCM

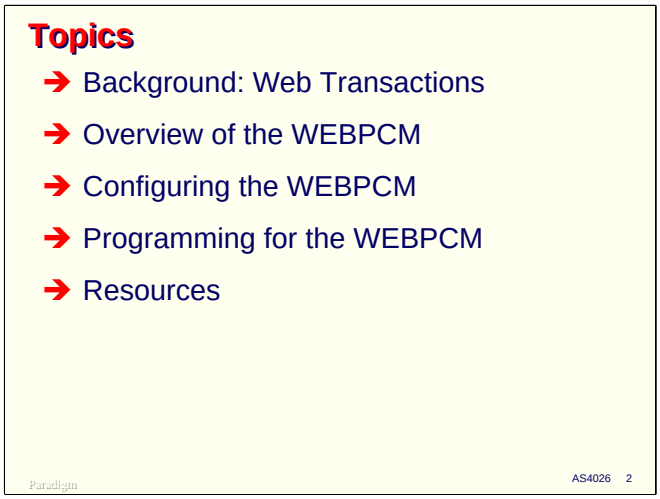
2002 UNITE Conference, Baltimore, Maryland.

Paul Kimpel

Paradigm Corporation
San Diego, California

http://www.digm.com

e-mail: paul.kimpel@digm.com



This presentation discusses the WEBPCM facility and its use under the Unisys ClearPath MCP.

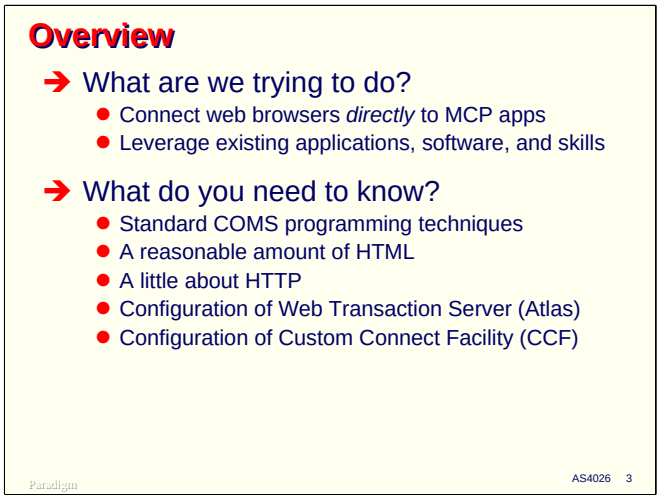
We'll begin with some background information on how web browsers and servers exchange messages, since it helps to know the basics of this when writing programs that process HTTP requests and compose HTTP/HTML responses. We'll also discuss HTML forms briefly, since they are very important for most business applications on the web.

Then we'll discuss the general features of the WEBPCM product, and how it relates to the Web Transaction Server (Atlas), Custom Connect Facility (CCF), and the MCP Transaction Server (COMS) products.

Next we'll see how to configure Atlas and CCF to support a WEBPCM application program.

The final piece necessary to implement a WEBPCM application, which we'll cover at a high level, is the application programming interface that COMS programs must use. This discussion will concentrate on using the WEBPCM with COBOL-74 or -85, but it applies as well to programming in the other languages supported by the WEBPCM.

Finally, we'll briefly look at a couple of sample WEBPCM programs in action and discuss where you can find more information and sample programs to guide your own work.



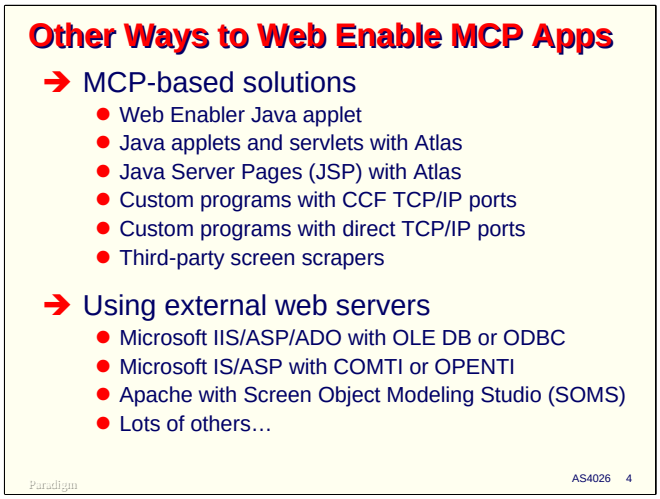
What are we trying to do with the WEBPCM and why should we use it?

The short answer is that it is one of the ways you can connect web browsers (clients) directly to MCP applications. This can be done over the global Internet, a private intranet, an extranet, or some combination of all three.

One of the big advantages of the WEBPCM is that it is largely based on tools and techniques that many MCP programmers already know. If you have experience writing transaction processing programs in Algol or one of the COBOL dialects for COMS direct window or remote file programs, you already know much of what you need in order to do web-based programming.

There are some significant additional skills that you will need however:

- You have to know a reasonable amount about Hypertext Markup Language (HTML). Even if you use an HTML editor to compose web pages, you should have a working familiarity with the basic tags and attributes. Most business interfaces seem to revolve around data entry forms and tabular presentations of data, so a good grounding in HTML forms and tables is almost essential.
- You should know a little about the Hypertext Transport Protocol (HTTP), which is the mechanism that web browsers and servers use to communicate between each other. The overview in this presentation, plus examination of the example programs, will tell you most of what you need to know in this area for basic web processing.
- Implementing WEBPCM applications involves changes to the Atlas web server configuration, so either you need to know this area or have access to someone at your site who does.
- Similarly, implementing these applications involves changes to the CCF configuration, so either you need to know how to do this or have access to someone who does.

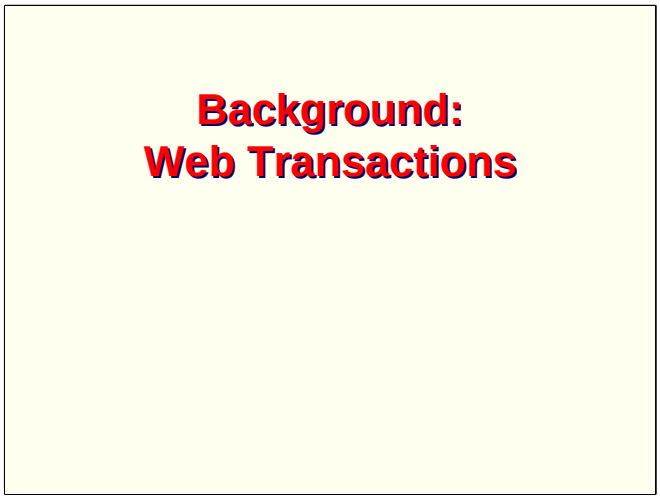


Before launching into more details about the WEBPCM, you should know that this is just one of the ways that you can web-enable an MCP application. There are a number of other approaches that use standard, bundled MCP software products:

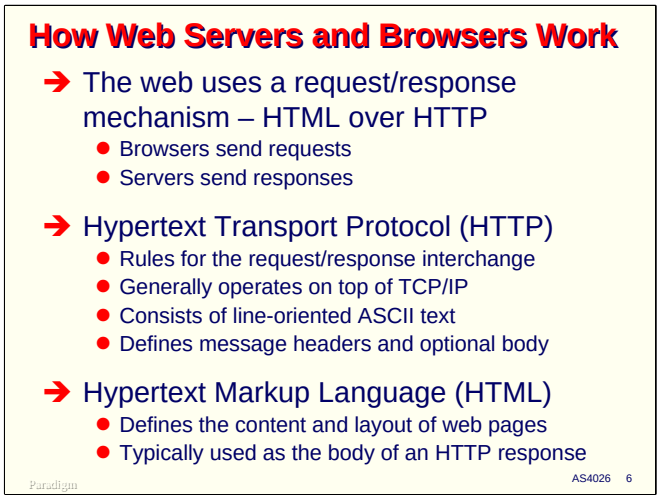
- The Web Enabler is a Java applet that implements a basic T27 terminal emulator. It allows an end user to access COMS, CANDE, and other MCSes in the MCP environment over the Internet using only a Java-enabled browser.
- The Atlas Web Transaction Server supports Java servlets. These are a server-side companion to applets, and provide a Java-based transaction processing facility in the MCP environment.
- Atlas also supports Java Server Pages, another way of generating dynamic content from the MCP Java environment.
- The Custom Connect Facility (CCF) provides an easy means to connect programs running in the COMS Transaction Server environment to standard TCP/IP ports. This is an excellent way to communicate between COMS programs and "intelligent" network agents, such as Visual Basic programs or custom controllers.
- You can also communicate with TCP/IP ports directly in user programs, through either port files or the relatively new Socket Library. This approach is more technically challenging than using CCF, but it offers more flexibility, functionality, and efficiency in return.
- Finally, there are a number of third-party products that will convert COMS/T27 data streams to HTML/browser interfaces.

There are also a number of approaches that are based on external web servers. The most popular approaches tend to use web servers running on the Windows side of a ClearPath system, viz,

- Microsoft IIS Active Server Pages (ASP) using ADO for data base services, and connecting to either OLE DB or ODBC in the MCP environment to access DMSII data bases.
- Microsoft IIS ASP using either COMTI to access COMS transaction streams or OPENTI to access the MCP Distributed Transaction Processing (OLTP/DTP) product.
- The new Screen Object Modeling Studio (SOMS) from Unisys, which interfaces to the Apache web server and can translate traditional "green-screen" transactions to a number of more modern representations.



To place the discussion of the WEBPCM and its facilities in context, we'll start by looking briefly at how web browsers and servers exchange messages.



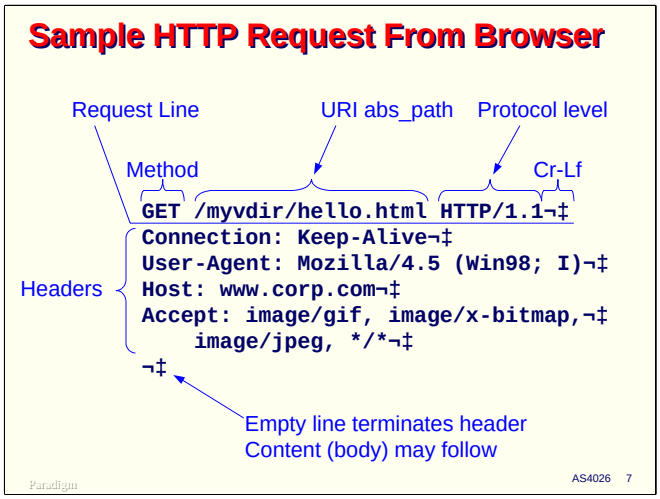
The basic idea of the World Wide Web is very simple: a browser (or client system – what the standards term a "user agent") sends a request to a web server and the server sends a response in reply. The details can get a lot more involved, since requests can be in many different forms, and responses can contain entities, such as hyperlinks and images, which cause the browser to automatically make further requests. A typical web page has many such embedded entities, so what to us is often just a simple entry of a URL or a click on a link, actually results in a flurry of request/response interactions between the browser and one or more servers to load all of the elements of the page.

The way this request/response interaction works is determined by the Hypertext Transport Protocol, or HTTP. Like most communications protocols, it consists of a set of rules that describe how messages are to be formatted and delivered. HTTP messages are generally composed as a stream of ASCII text, and are transmitted on top of the standard TCP/IP protocol. HTTP normally uses TCP port 80.

HTTP messages consist of two parts, as we will see shortly: a number of introductory lines called "headers", optionally followed by either text or binary data that makes up the "content", or body, of the message.

One of the most common types of data that makes up the content of a message is Hypertext Markup Language, or HTML. It typically consists of ASCII text, and is sent from servers to browsers to describe the response and how it should be rendered for the user. Most of us think of HTML as a way to generate a visual display of data, but it also has facilities to support non-visual renderings, especially for the vision-impaired.

The idea of non-visual rendering has been abstracted over the past few years into a new form, Extensible Markup Language, or XML, which is primarily designed as a vehicle for direct machine-to-machine transfer of data. Another variant, XHTML, is a redefinition of the facilities offered by HTML using an XML-compliant syntax.



What does an HTTP message look like?

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 called the "request line." It contains three tokens:

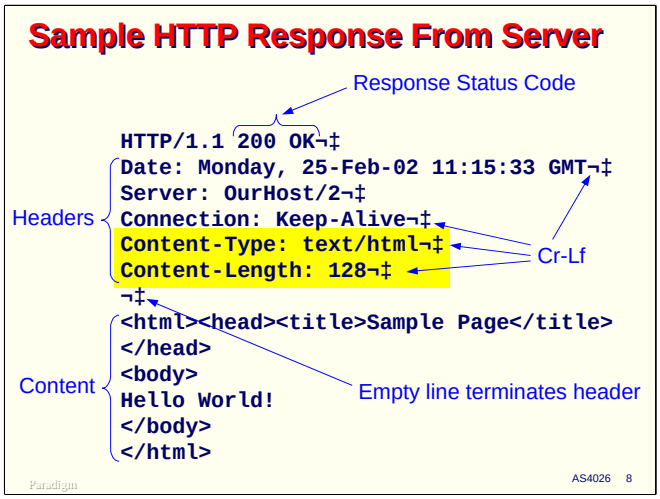
- The Method. This is essentially an HTTP transaction code, and determines the type of message. The most common methods are GET and POST.
- The Uniform Resource Locator (URI) absolute path. This identifies the "resource" on the server that the client is trying to access. It may map to a file in the server's 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 name/value pairs which 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, separated by a colon. Headers convey information about the message and instructions for operation of the protocol. You can access these headers inside application programs, and some of them are quite useful. 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-Length header indicating its size.



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

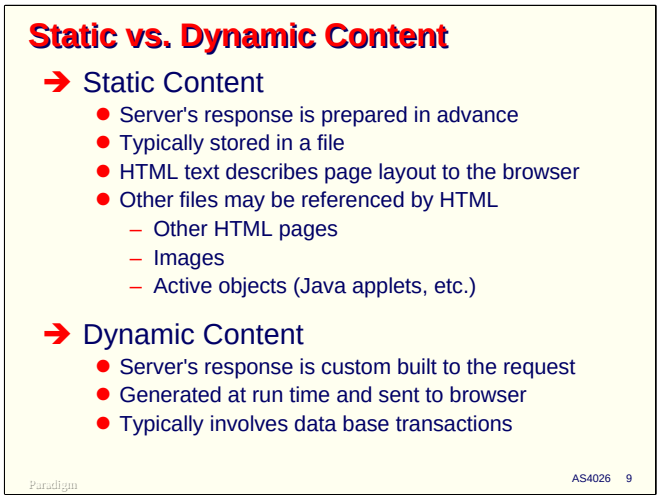
The first line indicates the level of protocol the server is using and a status code. This status code consists of two parts:

- 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, and the WEBPCM allows you to supply your own text on a case-by-case basis. 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.
  - 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 message. Most response messages 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-Length header, which tells the receiver how many octets (bytes) of body to look for.



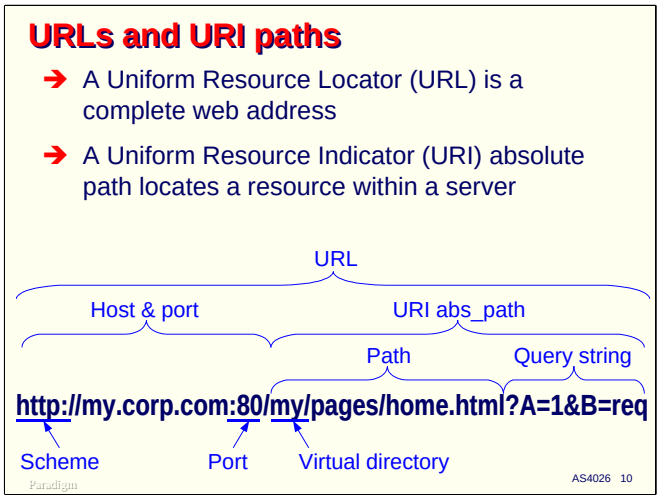
Web servers typically respond to browser requests in two fundamentally different ways.

The first, and probably most common, is static content. This is what got the web started, and is still a very important part of it. With static content, the response is pre-built and stored on the server, usually in a file. In response to a request that identifies a static resource, the web server simply wraps the data in an HTTP envelope and pumps it the down the network to the client.

These files are often simply HTML text which describe the content and layout for a page. The HTML text can contain links to other HTML pages and references to embedded objects – such as images and Java applets – that the browser will automatically load and render in the page.

The second type of content is dynamic. In this case, the server generates a custom response – typically a page of HTML text – dynamically from parameters in the request. Often the web server assembles the response based on transactions against a data base, in much the same way that traditional "green-screen" applications work.

Business applications with a web interface often involve a combination of static and dynamic responses from web servers, but it's the dynamic content that is critical. Generation and delivery of dynamic content is the primary concern of the WEBPCM, and is the form we will concentrate on most during this presentation.



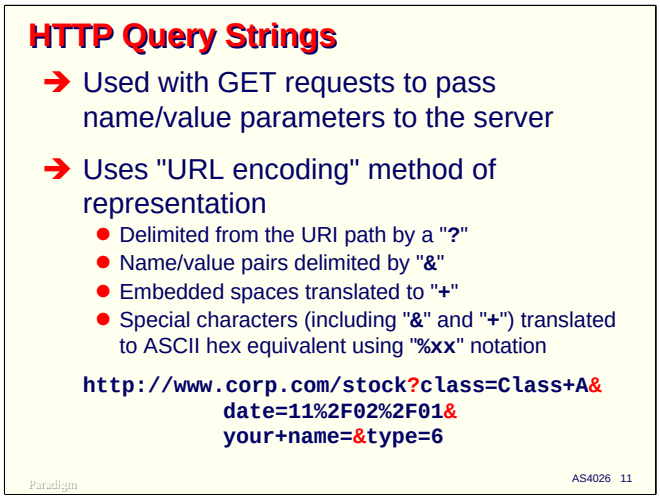
One of the most important concepts in configuring and programming for web server environments is that of addressing web resources. The mechanism for this is termed a Uniform Resource Locator (URL), which has as a component the Uniform Resource Indicator (URI) path. These are defined in RFC 2396.

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 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 which generates the content. The manner in which the query string is encoded is discussed in more detail on the next slide.

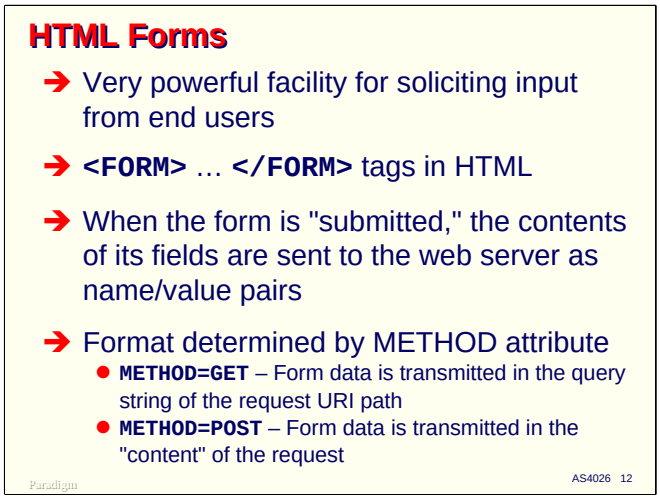


Query strings typically appear in HTTP requests using the GET method for a resource that generates dynamic response content. They pass parameters or other control information to the resource. Because the parts of an HTTP request line are delimited by spaces, URLs and URI paths cannot contain literal 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 which are reserved, or ones which 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 also be encoded using this method.



There are two HTML facilities which tend to be used extensively in business applications: tables and forms. Because HTML forms affect the way that data is formatted by the browser and sent to the sever, it's useful to see how they work.

Forms are defined in an HTML page between <form> and </form> tags. Within the form, you can include most other HTML entities, such as text, images, hyperlinks, but not another form. Most importantly, you can include some special tags which create basic GUI controls on the displayed page. These controls allow the end user to enter or select data values for the form.

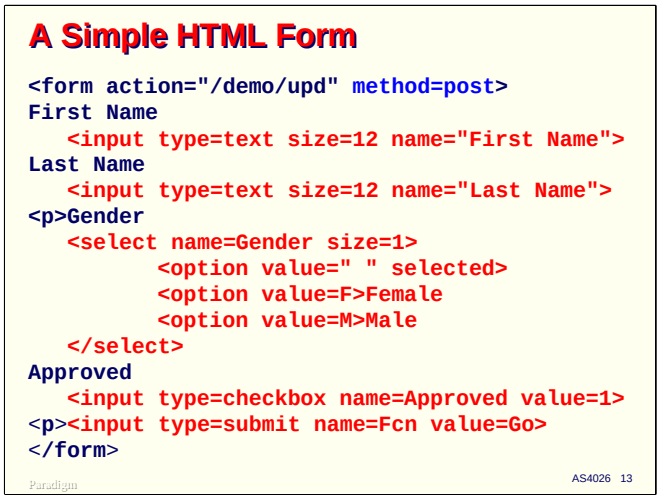
One or more elements on the form allow the user to "submit" the form, i.e., send it to the web server. When the user activates on of these controls, the contents of all the controls are formatted as name/value pairs and sent to the sever. The manner in which they are sent depends on the Method attribute of the Form tag:

- If METHOD=GET, the data from the form controls is URL encoded and sent in the HTTP request message as the query string.
- If METHOD=POST, the data is again URL encoded, but is sent as the body, or content, of the request message.

There are a variety of form controls you can specify in HTML. The most common one is the <input> tag. It supports several types of single-valued controls, including text fields, buttons, checkboxes, radio buttons, and a file-open dialog for uploading complete files from the client system to the server.

Another commonly used control is <select>, which allows you to define pull-down lists.

HTML 4.0 also supports some additional controls, namely, <textarea>, <option>, <optgroup>, <button>, <label>, and <isindex>

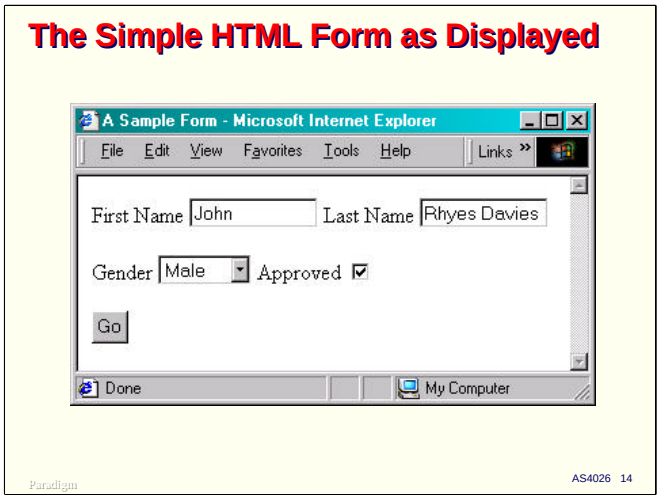


Here is an example of a simple HTML form. The "action" attribute in the form tag specifies the URL where the data from the form is to be sent when it is submitted. Form data is typically sent back to the same server which supplied the form to the browser, but this is not necessarily the case. Note that in this form, the data will be sent using the POST method, meaning it will appear in the content of the request instead of the query string.

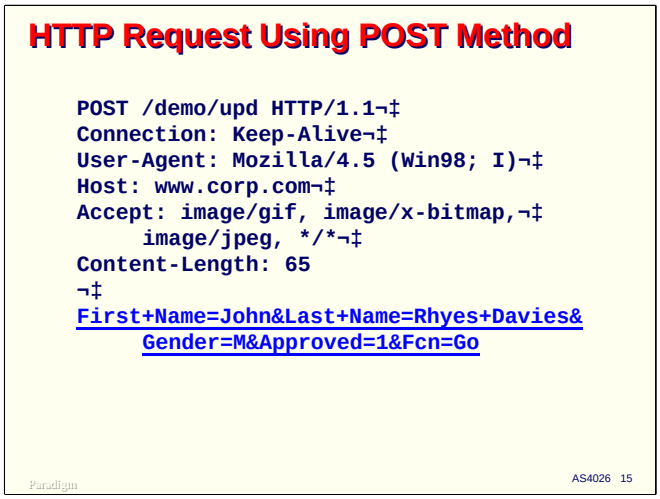
The first two <input> tags describe simple text boxes on the page. The <select> tag defines a pull-down list with three items. Each item in the list is defined by one of the <option> tags embedded between the <select> and </select> tags. The third <input> tag defines a simple checkbox. It will send a value of "1" if it is checked, and no value at all if unchecked.

The "name" attribute on each of the control tags will be used to construct the name/value pairs in the resulting request message for this form.

The final <input> tag defines a submit button. When the user activates this button, the data from all the form controls will be sent to the URL specified in the <form> tag.



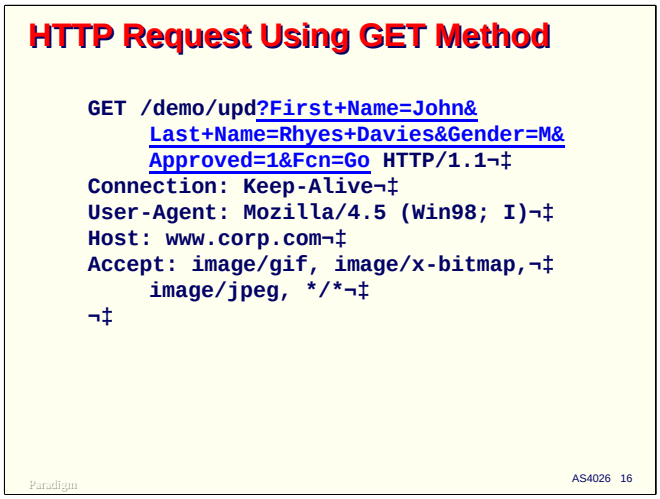
Here is the form from the HTML text on the prior slide, as it would be displayed in Microsoft Internet Explorer. The fields have been filled in, and the form is ready to submit.



When the completed form on the prior slide is submitted, the destination web server will receive an HTTP request message that looks something like this. Note the POST method in the request line, along with the URI path from the form's "action" attribute.

The data from the form controls appears in the body, or content, of the message. Note how the names and values of each of the controls has been translated using URL encoding. Also note the Content-Length header, which informs the server of the content's length.

As you might expect, parsing and decoding the headers and URL-encoded body is not a trivial task, especially in programming languages like COBOL. As we will see later, the WEBPCM provides facilities that will conveniently extract the data from this format and make it available to an MCP-based program.



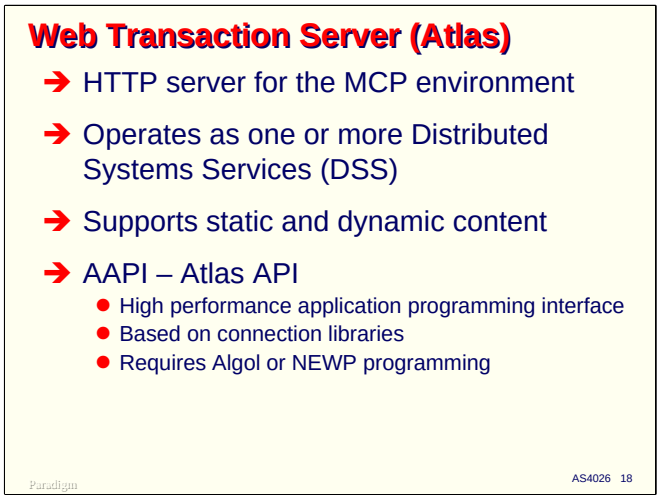
If the Method attribute in the <form> tag had been GET instead of POST, the submitted form would arrive at the server in a request message that looks something like this. Note that the URL-encoded name/value pairs are part of the request line instead of in the body of the message.

It is up to the designer of the form and the programmer on the server to agree on whether form data should be submitted using GET or POST. There are a couple of considerations to keep in mind, however:

- Most web browsers 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 which will send a large amount of data, it may overflow your browser or server capacity.
- Most browsers have the ability to display the current URL on the user's screen in a text box which can be edited and resubmitted. If this feature is enabled, the data from forms submitted using the GET method will also appear there, and potentially could be altered and resubmitted by the user. If you don't want the user to see or change the submitted data once it is sent from the form, POST is the better choice of method.

Overview of the WEBPCM

Let us now look specifically at the WEBPCM as it is implemented for the MCP.



Using the WEBPCM requires that you also use the Web Transaction Server for the MCP, more popularly known as "Atlas."

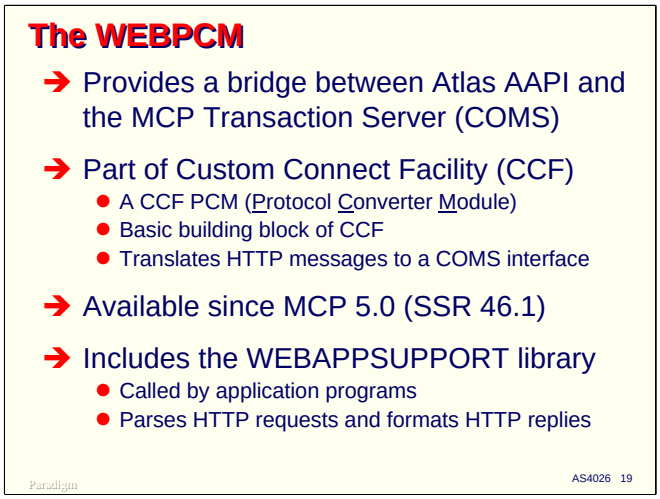
Atlas is a fairly complete HTTP server for the ClearPath MCP environment. It operates as one or more Distributed Systems Services, or DSSes. In it's default configuration, Atlas consists of two DSS providers, ATLASSUPPORT, which is available for normal web services, and ATLASADMIN, which is used to configure the Atlas environment and provide a platform for several Unisys sample programs, including those for the WEBPCM.

Atlas supports delivery of both static and dynamic content. Static resources, such as HTML pages and images, are stored as ordinary files in the MCP file system. Virtual directories in URI paths map to actual disk directories in the file system, with the rest of the path identifying a specific file under that directory.

Atlas has two mechanisms to support dynamic content. There is a "CGI" interface which exists only to provide compatibility with the old NX/Web Server product that was originally released with the ClearPath line. The interface of choice, however, is AAPI, the Atlas Application Programming Interface.

AAPI is a capable and high performance application program interface. It is based, however, on the use of connection libraries, which are supported only in the Algol and NEWP programming languages. Since most business applications are written primarily in COBOL, this originally made Atlas difficult to integrate into existing applications using existing staff skills.

Something else is needed to provide an interface between Atlas and the more traditional programming environments used at most sites. This is the role the WEBPCM attempts to fill.

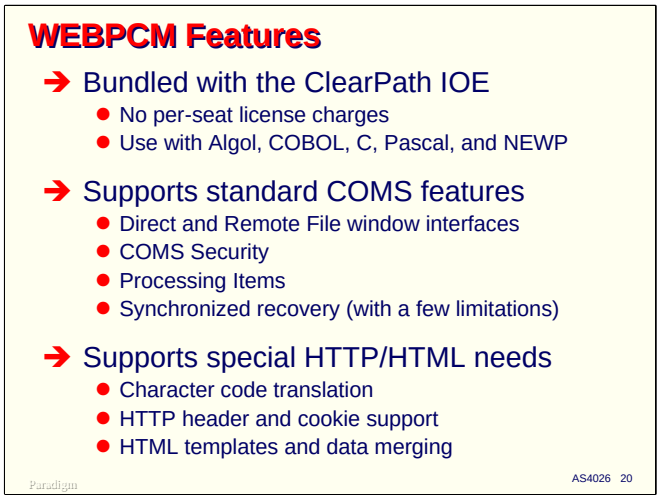


The purpose of the WEBPCM is to provide a bridge between the Atlas AAPI and the MCP Transaction Server, which most of us still insist on calling COMS. By bridging web transactions to the standard transaction server for the MCP environment, the WEBPCM allows you to bring forward a large set of existing skills for implementing web-based interfaces. It also allows you to program in most of the languages available for the MCP.

The WEBPCM is part of the Custom Connect Facility (CCF) product. It is implemented as a CCF Protocol Converter Module, or PCM – hence the name. PCMs are the basic building block of CCF. Their purpose is to translate external communications protocols and interfaces to COMS interfaces. The primary purpose of the WEBPCM is to translate between HTTP messages for Atlas and standard COMS messages.

The WEBPCM has been available since MCP 5.0 (SSR 46.1), but can work with MCP 4.0 as well.

From a programmer's perspective, the most visible part of the WEBPCM is a library called WEBAPPSUPPORT. This library exports a number of routines that interface COMS programs to the WEBPCM and provide facilities for parsing HTTP requests and constructing HTTP responses. We will spend considerable time later in this presentation talking about the facilities of the WEBAPPSUPPORT library.



The WEBPCM is bundled with the standard ClearPath Integrated Operating Environment. If you are running on MCP 5.0 or later, you have everything you need to interface your MCP system to web browsers.

There are no per-seat license charges associated with the WEBPCM. You can potentially support as many simultaneous web users as the size and capacity of your ClearPath system will handle. This is an especially important consideration for sites with LX systems. You can program for the WEBPCM in most of the common MCP languages, including Algol, COBOL-74, COBOL-85, C, Pascal83, and NEWP.

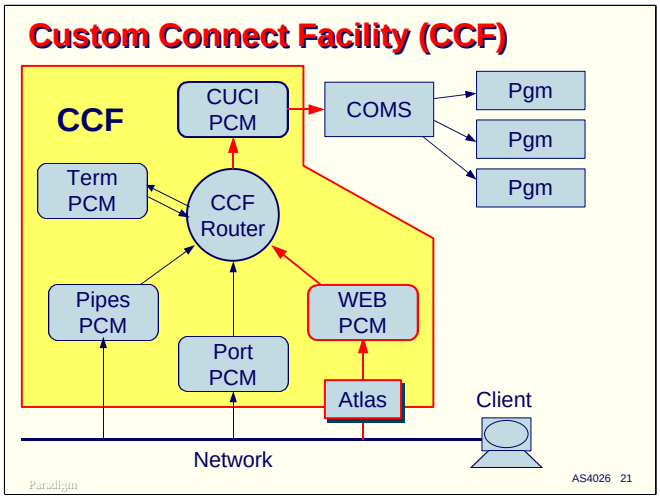
The WEBPCM supports all of the standard features of COMS, including

- Both Direct and Remote File Window interfaces.
- COMS station, user, window, and trancode security.
- Processing items.
- DMSII synchronized recovery. There are a few WEBAPPSUPPORT calls which cannot be used when synchronized recovery is configured, but these are usually not critical features.

Most importantly, the WEBPCM provides convenient facilities to ease the task of parsing and constructing HTTP messages. It can translate between the EBCDIC code typically used in MCP applications and any number of national character sets, including ASCII, used by browsers. It also provides very good support for extracting values from HTTP headers, cookies, and URL-encoded fields of query strings and message content.

The WEBPCM also provides good support for formatting output messages. It can add custom headers to a response, construct cookies, and place arbitrary text or other data in the response message body.

One very nice feature of the WEBPCM is its ability to merge data values with an HTML template to produce a custom string of formatted HTML. This is especially useful when programming in COBOL, or when HTML authoring and MCP applications programming are done by different members of your staff.

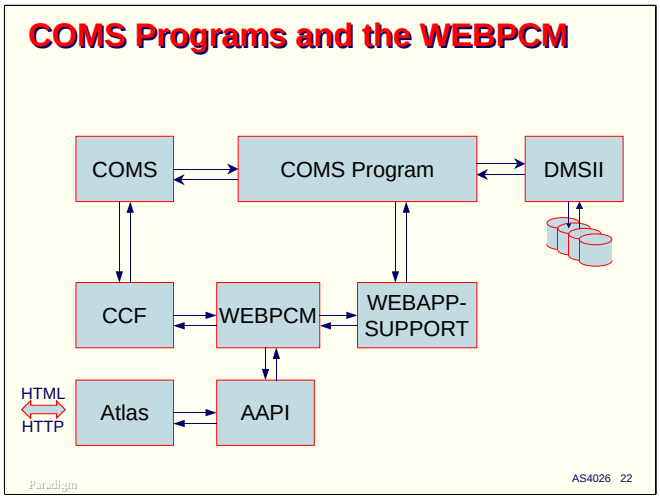


The WEBPCM is part of the Custom Connect Facility and uses other portions of CCF to interface with COMS.

The Custom Connect Facility consists of a number of Protocol Convert Modules and a Router module that connects them and passes messages between them. The ultimate destination of most messages passing through CCF is the CUCI PCM. CUCI is the COMS Universal Connection Interface, a mechanism for creating virtual stations and linking them to the COMS environment. The job of the CUCI PCM is to interface the rest of CCF to COMS.

The standard CCF product from Unisys includes modules for named pipes and TCP/IP ports. These are used by NX/View and the Web Enabler applet, but sites can configure additional interfaces to them as well. The TERM PCM does not connect directly to the outside world, but instead provides virtual terminal services to the named pipe and TCP/IP port PCMs.

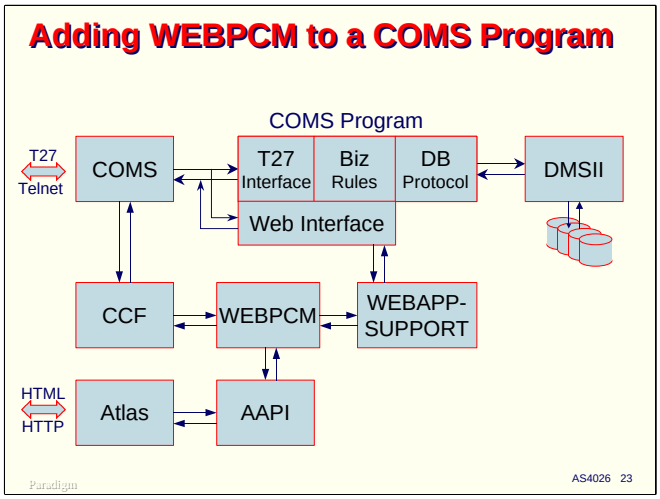
As shown, the WEBPCM sits between Atlas and the CCF router. It communicates with Atlas using the AAPI, routes the web traffic to the CUCI PCM, which represents it to COMS as messages from one or more virtual stations. COMS is not aware that the messages came from the web server, and routes them as it would any other incoming traffic to the appropriate windows and programs. The WEBPCM routes only to the CUCI PCM. It does not use the TERM PCM.



An MCP-based transaction processing program typically interfaces to COMS on one side, and executes transactions against a data base on the other. This arrangement does not change when using the WEBPCM. All of the features of COMS are still available, including Trancode routing, security, and Processing Items.

For messages originating from the web, the program receives the data via the WEBPCM and COMS in a special format called a Message Object. The program interfaces with COMS in the usual manner, but makes calls on the WEBAPPSUPPORT library to retrieve header and content data from the message object and construct a reply within it.

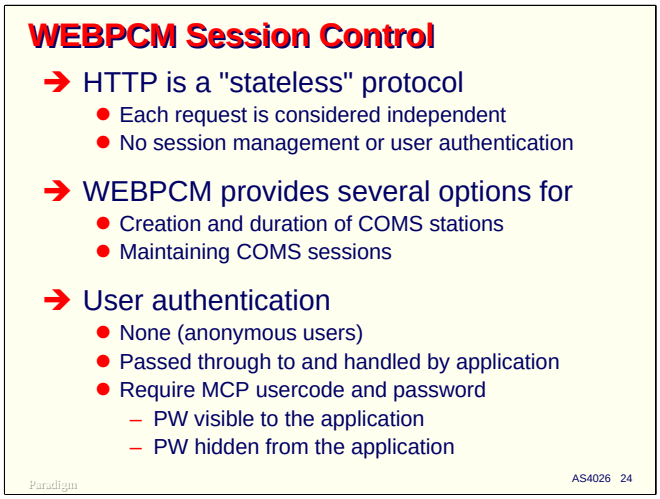
Thus, with the WEBPCM, the overall flow of data and structure of COMS programs is the same as with traditional terminal interactions, but the details of parsing input messages and formatting output messages changes quite a bit.



Most programs for the WEBPCM will probably end up being writing from scratch, but there is no reason why you cannot add web transaction capability to an existing COMS program. The program can detect internally whether the message came from the WEBPCM, and process the message accordingly.

Since the format of data from the web is considerably different than that from traditional terminal screens, you would probably need to add code to an existing program for input editing and output formatting, but the business rules and data base interface for the application could remain unchanged.

If you are very clever, you could even write COMS processing items that would do the necessary parsing and formatting for the web transactions outside of the program, and pass to the program messages that look just like those coming from a standard terminal emulator. This is not a trivial task, however, and would restrict you to building web interfaces that merely echo those of your existing green-screen interfaces.



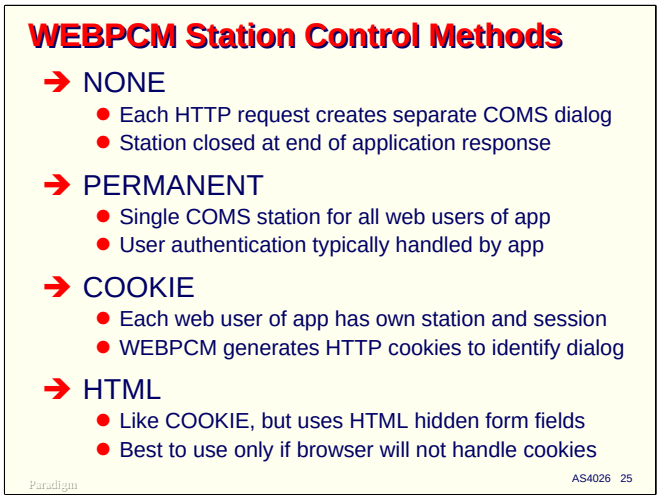
One of the major issues in dealing with web transactions is that HTTP is a "stateless" protocol. This means that each request from an end user is treated as a separate, independent entity. As far as HTTP is concerned, you don't "log on" to a web server, you just send it a request and it sends a response. There is no concept of a session or of user authentication.

In many situations, especially those involving static content, this is exactly what you want. You don't care who the users are, or what they sent you a few minutes ago. In many business applications, however, stateless, unauthorized access is not acceptable. The end user must be authenticated, and their interactions with the server must take place within the context of a session.

To support this, the WEBPCM provides several options for creating COMS sessions and controlling their duration. Since COMS session are associated with COMS stations, these options also control how the virtual CCF stations get created and how long they exist.

The WEBPCM also provides three primary options for authenticating users.

- The simplest is no authorization at all. You can use the WEBPCM with anonymous users and simply not have per-user sessions.
- User authentication can be handled by the COMS application. This is typically the best method when the users are identified within the application and not in the MCP's USERDATAFILE. It is relatively easy, using HTTP headers and cookies, to request a user identifier and password from the browser and maintain session state yourself.
- Finally, you can have the WEBPCM authenticate the end user against the MCP USERDATAFILE. When using this option, the MCP usercode is visible to the application. One of the options you configure in WEBPCM is whether the application can see the password the user enters.

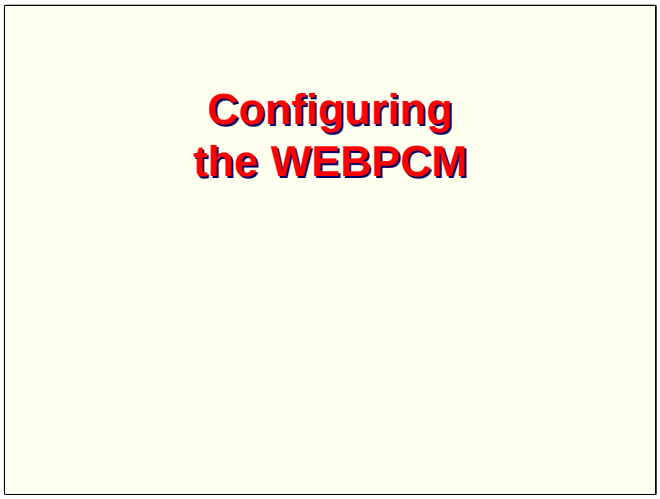


The WEBPCM also provides a number of options for station and session management.

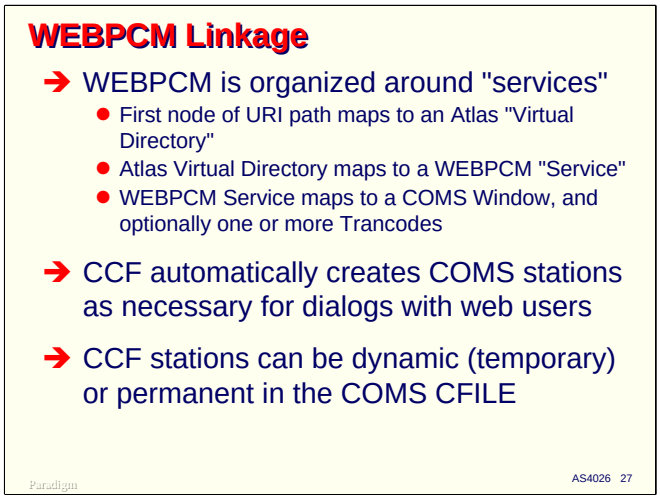
- The most basic form of session management is none at all. With this option, each web transaction is treated as a separate, independent session. CCF creates a virtual station when the request arrives, establishes a COMS session, and passes the message to COMS for routing to a program. After the response is completed, the session is terminated and the station is closed. While easy to set up and use, this option has a lot of overhead.
- The second option is a single, permanent station. With this option, CCF creates a virtual station and COMS session for the WEBPCM application and routes all users through it. If per-user authentication is required, or per-session state must be maintained, the application must do that itself. The station and session are created automatically when the first user submits a message, and continue to exist until closed manually or the WEBPCM terminates. This approach is very efficient, but often requires considerable programming within the application if your situation requires you to maintain session state yourself.
- The WEBPCM offers two methods for creating virtual stations and COMS sessions for each end user. The first, and simplest to use involves HTTP cookies. The WEBPCM will automatically request a usercode and password from the end user, authenticate it against the MCP USERDATAFILE, and create a cookie that identifies the session. When the browser sends its next request, the cookie is automatically included in the HTTP message, which the WEBPCM uses to route the message through the appropriate virtual station. The result is a COMS environment very much like you have with Telnet and terminal emulator clients.
- Some users have cookie support turned off in their browsers, so the WEBPCM provides an alternate to the cookie option that relies on hidden HTML form fields. Using this option requires cooperation from the application, since it must generate the HTML forms and place a WEBPCM session identifier in each outgoing response. This option is normally used only if cookies are not supported by your client systems.

The WEBPCM is configured as multiple "services," and each service can have different session management settings. Therefore, different applications on your system, or even different parts of the same application, can use different session management techniques.

The options for session management and user authentication interact, as we will see in the next section on configuring the WEBPCM.



Having looked at the basic capabilities of the WEBPCM, let us now see how it is configured in the MCP environment.



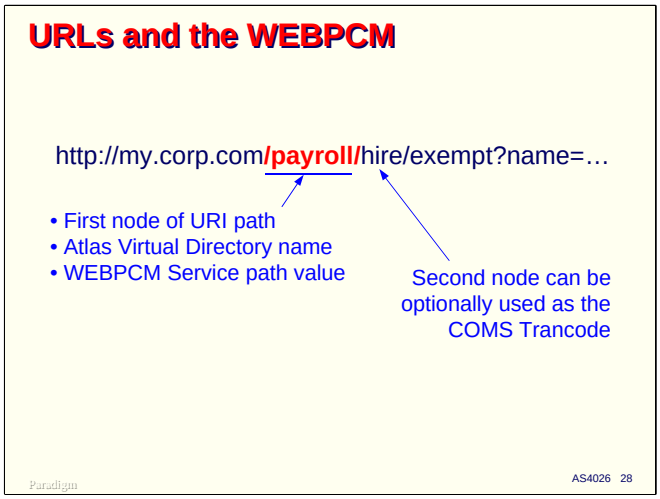
The WEBPCM is organized and configured around an entity called a "service." You can configure multiple services in the WEBPCM and assign different attributes to each one. Using multiple services allows you to route web traffic to multiple applications, or multiple programs within the same application.

An incoming HTTP request is associated with a service as follows:

- The first node of the URI absolute path in the request line is the "virtual directory."
- The name of this node must be defined in Atlas and associated with the WEBPCM.
- The WEBPCM service identifies the Atlas virtual directory it is to be associated with. In turn, it identifies a COMS Window, and optionally, a Trancode. COMS will use these to route the message to the appropriate program.

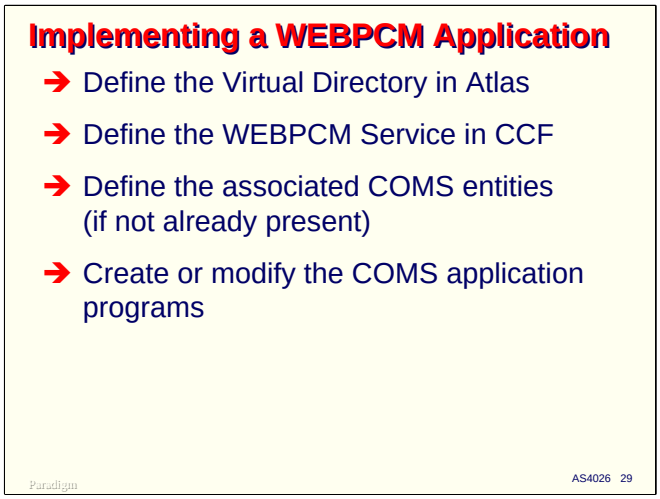
CCF will automatically create one or more virtual stations and corresponding COMS sessions for the WEBPCM service. The number of stations and manner in which they are created and maintained depend on the options for user authentication and station control that are configured for the service.

A feature that is very important for web environments which are accessed by the general public, especially when employing per-user sessions, is the DYNAMIC attribute of CCF stations. When this attribute has a value of True, the stations created by WEBPCM are temporary – they are removed from the COMS CFILE when they are closed. If you do not need to override the default attributes of WEBPCM stations, you can use this feature to prevent large numbers of web-originated stations from cluttering your CFILE.



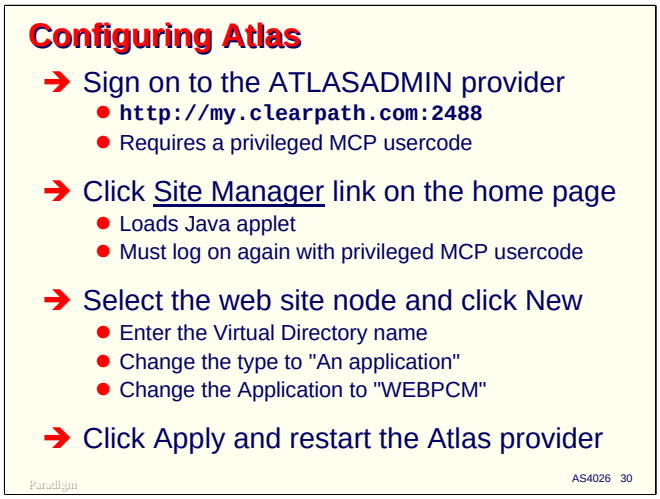
The first node of the path in a URL is critical. It identifies the Virtual Directory, which in turn selects the WEBPCM service.

As an option on the configuration of the WEBPCM service, the second node of the path can be passed through to COMS as the Trancode for the message. In any case, the entire path is visible to the COMS program, so it can use its value to perform internal routing or other activities.



To implement a WEBPCM application you need to do four things:

- Define the Virtual Directory in Atlas.
- Define the corresponding WEBPCM Service in CCF.
- If the web transactions will be going to new COMS entities, you will need to define the corresponding entities in COMS – Windows, Programs, Agendas, Trancodes, etc. If these already exist, then you simply need to know their names.
- Finally, you must create or modify the COMS program or programs that will handle the web transactions. We will discuss this area in more detail in the next section of the presentation.



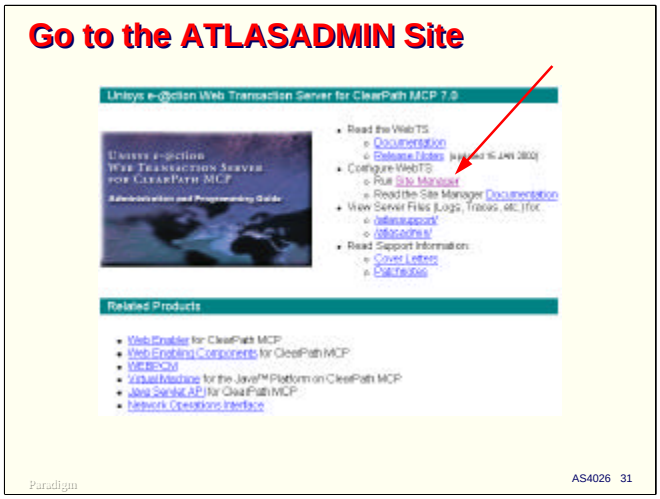
The first step in configuring a WEBPCM application is to configure the Virtual Directory in Atlas.

To do that you need to sign on to the administrative site for Atlas using a web browser. This site is configured by default on port 2488. Assuming your ClearPath has a domain name of nx.corp.com, open the following page in a browser:

```
http://nx.corp.com:2488
```

The first thing you will see is a sign-in dialog. You will need a privileged usercode and password to gain entry to the administrative site.

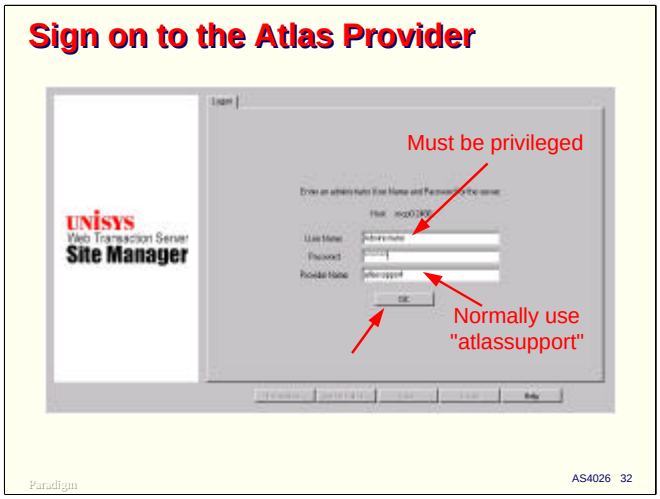
The remaining steps are illustrated on the following slides.



Once you successfully log in to the administrative site, Atlas should send you the site's home page.

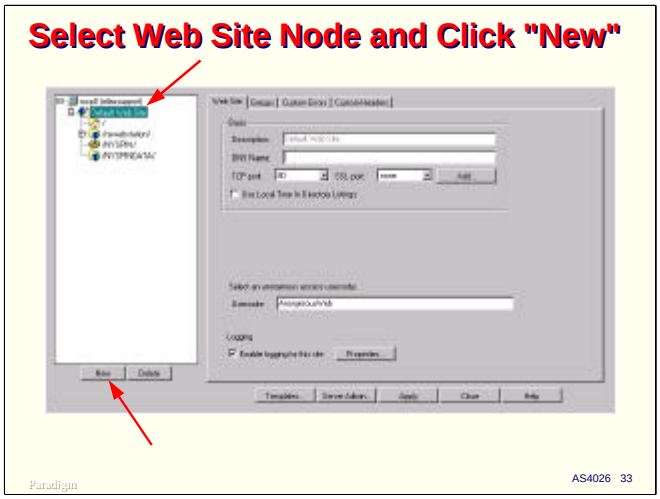
On the home page, click the **Site Manager** link.

Also note on this page that there are a number of links under "Related Products." One of these is for the WEBPCM, which will run the sample programs that are delivered with the product.



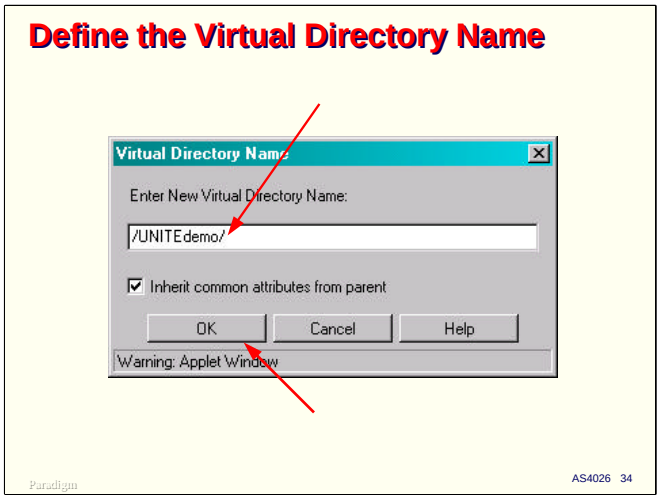
Clicking the Site Manager link will bring up the sign-in page for the Site Manager applet in a separate window. You must have Java installed in your browser to use this configuration tool. The applet normally takes a few seconds to load on a high-speed network, so you'll see a short delay. The delay will be longer over a dial-up connection.

- Enter a privileged MCP usercode and password again.
- You normally want to add WEBPCM services to the standard provider, so the Provider Name should be "`atlassupport`". You can also add services to the administrative site, "`atlasadmin`", or to an additional provider for your site. See the Atlas documentation for information on creating additional providers.



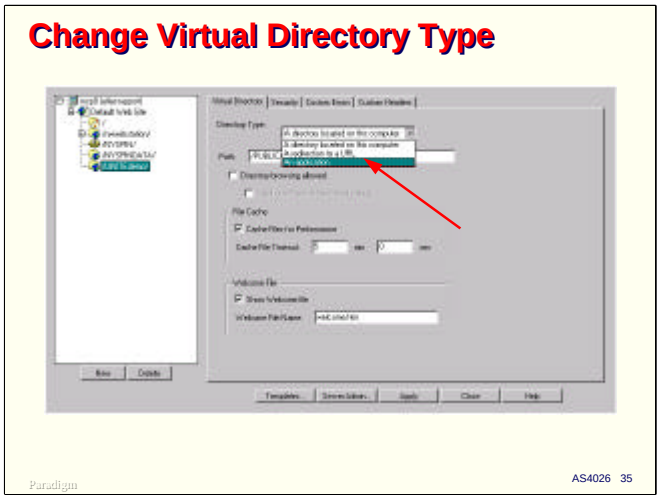
After logging in to the Site Manager for your chosen provider, it will display the root of its entity configuration tree.

- Click the "+" node to expand the tree to show the web sites configured for this provider (there will normally be only one, the default).
- Click on the web site's "+" node to expand it, then click on the name of the web site.
- Click the New button. This will start the process of configuring a new Virtual Directory.



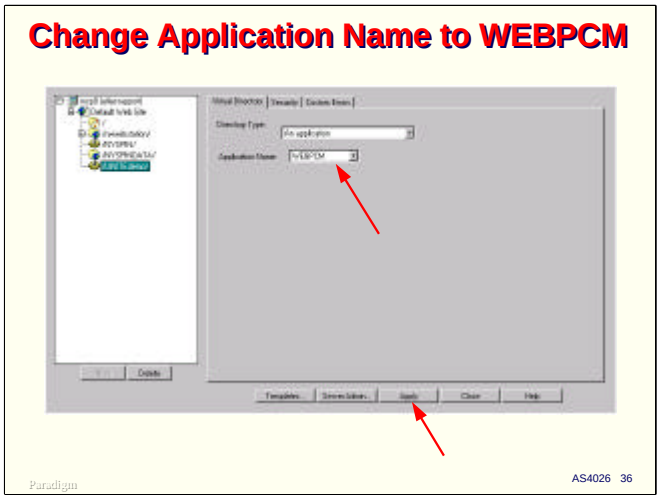
The Site Manager will display a dialog box for the name of the new Virtual Directory you are about to create.

- Enter the name you wish for the Virtual Directory.
- The name must start and end with a slash (/). It cannot contain slashes or spaces and must be less than 255 characters in length.
- The checkbox "Inherit common attributes from parent" should normally be checked.
- Click OK.



Once you define the Virtual Directory name, the Site Manager will add it to the configuration tree and display a panel with the directory's attributes. By default it creates a Virtual Directory that maps to an MCP file directory. You need to change this to map to the WEBPCM.

Simply pull down the Directory Type list and select "`An application`". The panel will change to show a different set of attributes.



The only attribute to configure for applications is the Application Name.

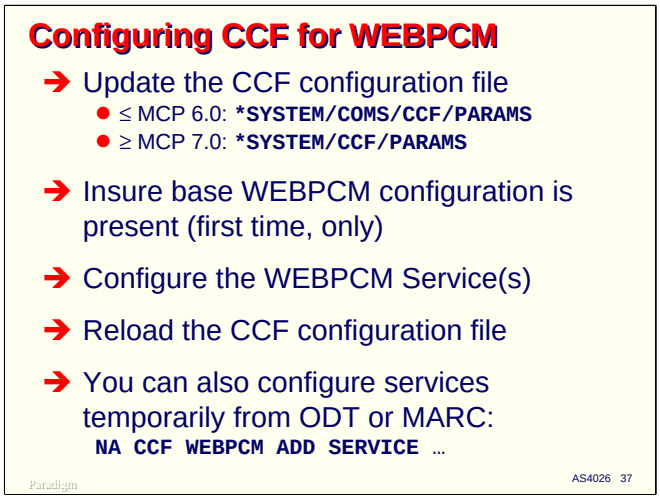
- Select WEBPCM from this list. Applications are defined elsewhere in the Site Manager. On many systems, WEBPCM may be the only one configured.
- If you have more Virtual Directories to configure, click the web site name, click New, and do them now.
- When you are finished adding Virtual Directories and making any other changes to the Atlas configuration, click the Apply button at the bottom of the panel.

When you click Apply, the Site Manager will display a dialog box warning you that the Atlas configuration for this provider will be updated and the previous configuration replaced. Click Yes on this dialog to proceed.

After writing the new configuration back to its file under the MCP, the Site Manager will prompt you to restart the Atlas provider. The changes will not take effect until the provider is restarted. You can choose a slow shutdown to allow currently open connections to complete, or a fast shutdown, which will terminate any open connections and restart the server immediately.

If you do not want to restart the provider just yet, you can click Cancel on the dialog box and restart the provider manually later.

This completes configuration for Atlas. The next step is to configure the WEBPCM service in CCF.



You modify the CCF configuration by making changes to its PARAMS file. This is a standard SEQDATA file which can be updated using any of the editing tools available under the MCP.

Note that in MCP 7.0 (SSR 48.1), CCF was separated from COMS and the naming convention for its files changed.

- Prior to 7.0, all CCF files begin with `*SYSTEM/COMS/CCF/=`
- Starting with 7.0, all CCF files begin with `*SYSTEM/CCF/=`

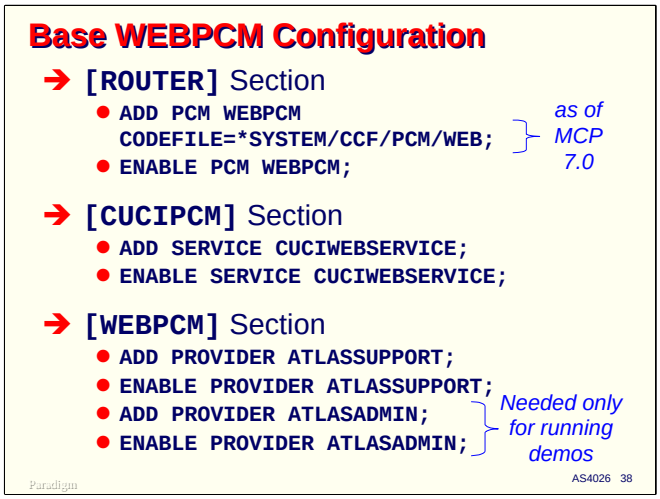
If you upgraded to MCP 7.0 using Simple Install or Install Center, these changes should have been applied automatically for you.

If you acquired your ClearPath system prior to MCP 5.0, you may want to assure that the base entries necessary to support the WEBPCM are in the CCF PARAMS file. The next slide details what entries must be present.

Once the base entries are present, you configure WEBPCM applications by simply adding the configuration for the their service to the PARAMS file and reloading it so the changes will take effect. Since CCF providers must be disabled in order to change their configuration, the easiest way to do this is by simply restarting CCF.

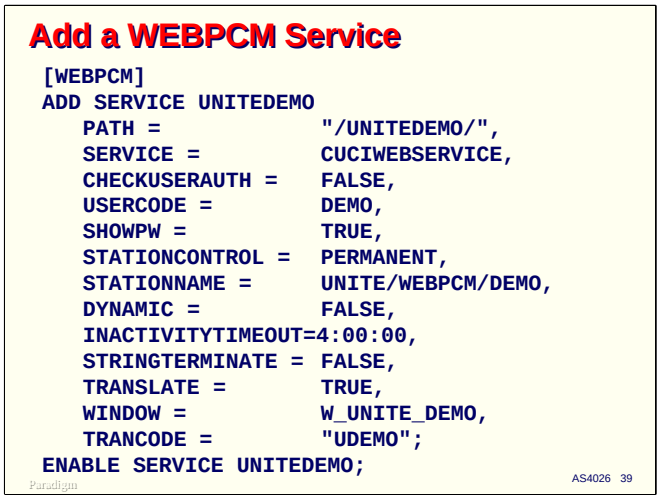
You can also modify the CCF configuration without shutting down CCF by entering NA CCF WEBPCM commands through the ODT or MARC. Modifications made this way are temporary, however, and are lost when CCF restarts.

The most effective way to make CCF configuration changes while the system is running is to enter temporary changes using NA commands, then update the PARAMS file so the changes will be loaded the next time that CCF starts. The downside to this approach is having to make the changes in two places, and remembering to do so.



There are entries in three sections of the CCF PARAMS file that must be present in order for the WEBPCM to work at all.

- In the **[ROUTER]** section, the WEBPCM provider must be defined and enabled.
  - On MCP 7.0, the name of the provider code file is `*SYSTEM/CCF/PCM/WEB`.
  - On earlier releases, the name is `*SYSTEM/COMS/CCF/PCM/WEB`.
- In the **[CUCIPCM]** section, you must define and enable the CUCI service that interfaces to the WEBPCM, CUCIWEBSERVICE. There are no attributes for this service.
- In the **[WEBPCM]** section, you must define and enable any Atlas DSS providers that will be used with the WEBPCM. Most sites can just use the two default ones, `ATLASSUPPORT` for normal web transactions, and `ATLASADMIN`, for Atlas administration. You only need the `ATLASADMIN` site in WEBPCM if you plan on running the demonstration programs that are released with the rest of the software. Note that in the default CCF configuration, the WEBPCM provider for `ATLASSUPPORT` is not enabled, so you will probably want to change that.

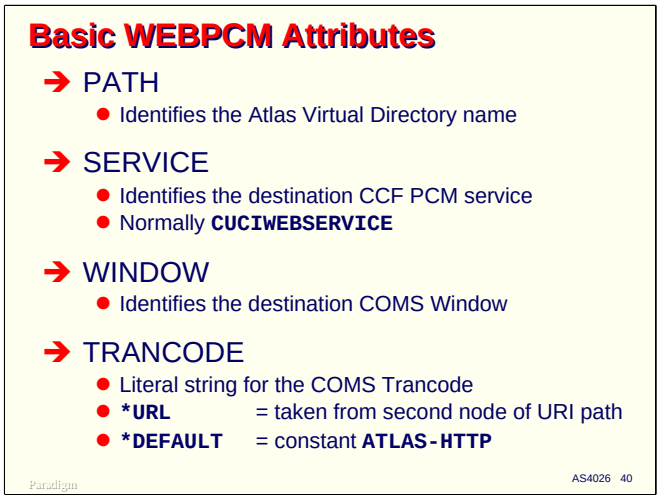


Once the base entries for the WEBPCM are present in the CCF PARAMS file, you can add WEBPCM services. These also go in the **[WEBPCM]** section of the PARAMS file.

The ADD SERVICE command creates a new WEBPCM service in the CCF configuration. The service is specified as a set of attributes. We will come back to this example and discuss the attribute settings shown here after we have covered the types of attributes available in the next several slides.

In addition to creating a service, you must enable it using the ENABLE SERVICE command before it can be used. In most cases you will want to place the enable command in the PARAMS file so the service will be enabled automatically when CCF initializes.

The WEBPCM also supports MODIFY, DELETE, and DISABLE commands for services. These are normally only used with NA commands through the ODT or MARC to make modifications to the running configuration.



There are four basic attributes that are always associated with a WEBPCM Service.

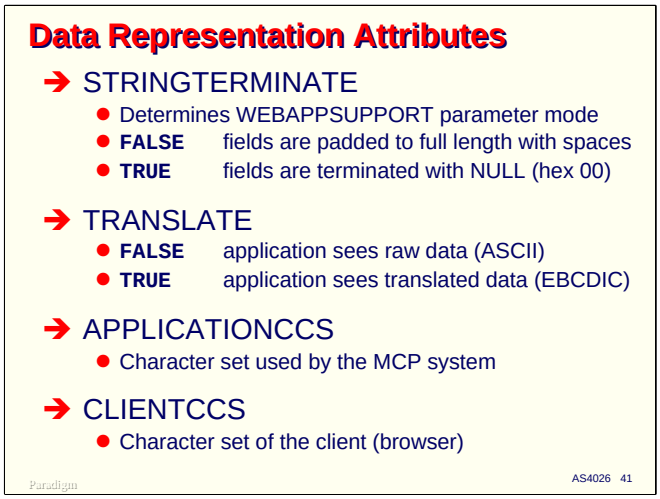
PATH identifies the name of the Atlas Virtual Directory associated with the Service. The value of this attribute must match the name, including the leading and trailing slashes, of an application Virtual Directory for the WEBPCM in the Atlas configuration.

SERVICE identifies the CUCI Service through which the HTTP messages will be routed to COMS. In a typical configuration, the name of this Service is `CUCIWEBSERVICE`.

WINDOW identifies the COMS Window to which the HTTP messages for this Service will be routed.

TRANCODE is optional. It supplies the value of the Trancode that will appear in the WEBPCM message object and (if being sent to a Direct Window) the Trancode which COMS will use to route the message to a program. This attribute can be specified in four ways:

- As a literal string. The value of the string will be passed as the Trancode to COMS.
- The keyword `*URL`. In this case, the Trancode will be taken from the second node of the URI path, immediately after the node that identifies the Virtual Directory.
- The keyword `*DEFAULT`. In this case, the WEBPCM will supply a default Trancode of `"ATLAS-HTTP"`.
- If the `TRANCODE` attribute is not specified, it is the same as specifying `*DEFAULT`.



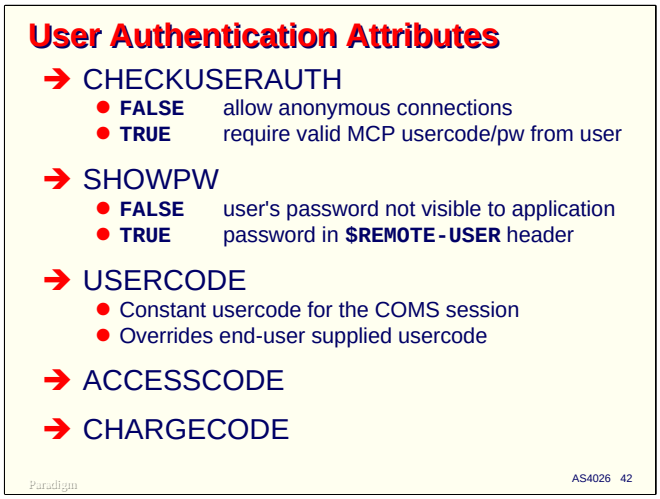
There are four Service attributes that determine how the data from HTTP requests will be presented to the application and how parameters to the WEBAPPSUPPORT routines will be formatted.

`STRINGTERMINATE`. If True, strings passed as parameters to WEBAPPSUPPORT routines will be terminated with a NULL byte (hex 00, COBOL `LOW-VALUE`). If False, strings will be padded with spaces out to their full length. The False setting is more convenient for COBOL programs, but incurs more overhead to pad the parameters with spaces.

`TRANSLATE`. If True, string parameters will be translated from the character code of the client (browser) system to that of the application (COMS program). By default, this translation is from ASCII to EBCDIC. The reverse translation will be performed for outgoing messages. If this attribute is False, no translation is done. For COBOL applications, the True setting is usually more desirable.

`APPLICATIONCCS`. If translation is in effect, this attribute names the Coded Character Set (CCS) that will be used by the application. The default is EBCDIC.

`CLIENTCCS`. If translation is in effect, this attribute names the Coded Character Set that will be used by the client system. The default is ASCII.



There are five WEBPCM Service attributes which control user authentication.

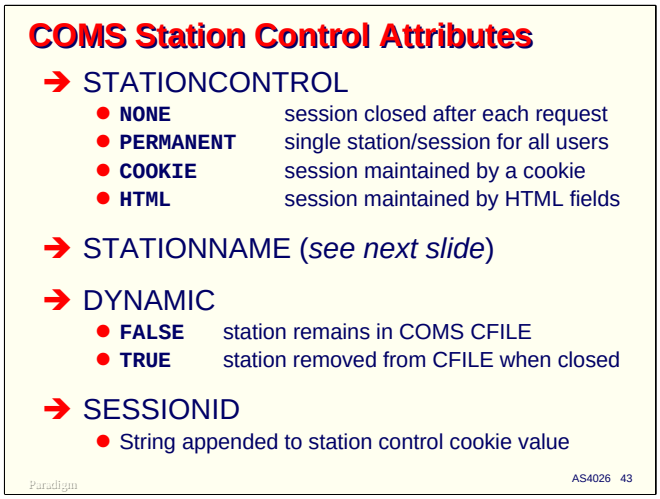
CHECKUSERAUTH. If this attribute is True, the WEBPCM will request a usercode and password from the end user and validate them against the MCP USERDATAFILE. This setting is normally used in conjunction with a STATIONCONTROL setting of COOKIE or HTML to create normal per-user COMS sessions. If this attribute is False, the WEBPCM performs no authentication, in which case the users will be anonymous, or the application must handle authentication itself.

SHOWPW. This attribute controls whether the application program can see the password a user enters from their browser. This should normally be False when CHECKUSERAUTH is True, and True otherwise. If the value of this attribute is True, the usercode and password entered by the user can be retrieved using the $REMOTE-USER pseudo-header. The usercode and password will be separated by a colon in the header value.

USERCODE. If CHECKUSERAUTH is False, you can use this attribute to supply a fixed usercode for the COMS session.

ACCESSCODE. This can be used with USERCODE to supply a fixed ACCESSCODE for the COMS session.

CHARGECODE. This can be used with USERCODE to supply a fixed CHARGECODE for the COMS session.



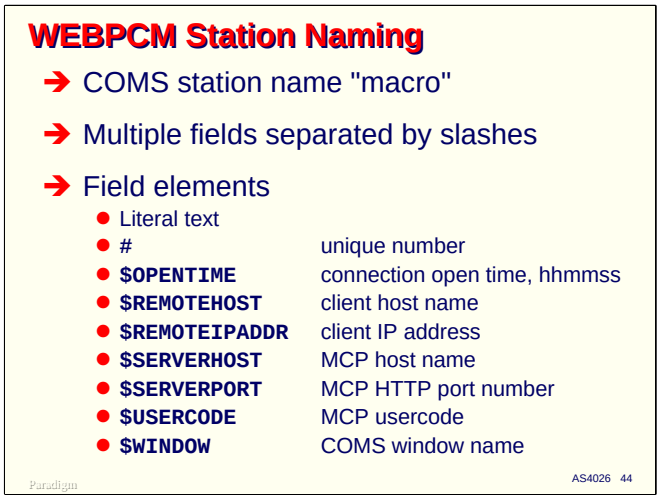
There are four Service attributes which control how COMS virtual stations and sessions are managed.

STATIONCONTROL. This attribute takes one of four values which describes the general method the WEBPCM is to use in managing COMS sessions, as discussed earlier in the overview.

STATIONNAME. This attribute defines the name that will be assigned to the virtual COMS station when it is created by CCF. It can have a constant value, or it can be constructed from a number of variables at run time. Since COMS cannot support more than one station at a time with the same name, constant station names are normally used only when `STATIONCONTROL` is `PERMANENT`.

DYNAMIC. If True, the COMS virtual station will be deleted from the CFILE when the station's CCF dialog is closed. If False, the station will remain in the CFILE.

SESSIONID. This string value is normally used only when `STATIONCONTROL` is `COOKIE` or `HTML`. The string value is appended to the value of the session's cookie.



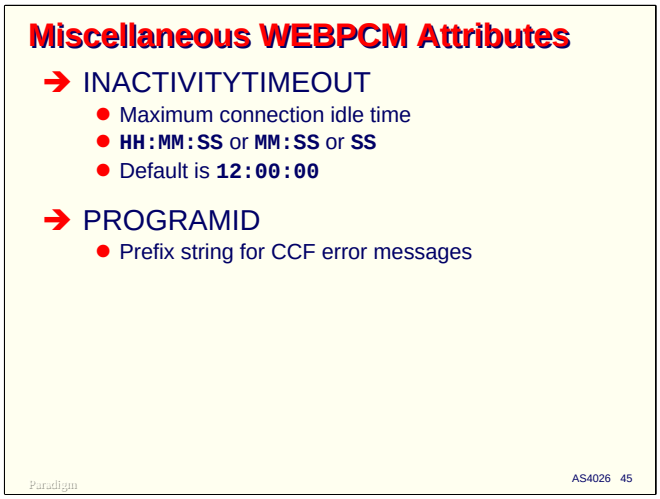
WEBPCM station names can be constructed at open time using a simple macro capability. The name can consist of multiple nodes separated by slashes. Each node can be one of the following:

- Literal text. The text will become the value of the node.
- **#**. The node will consist of an integer value chosen such that the resulting station name is unique.
- **$OPENTIME**. The time the station dialog was opened, in HHMMSS format.
- **$REMOTEHOST**. The first node of the remote client's domain name, if available.
- **$REMOTEIPADDR**. The remote client's IP address in decimal format with underscores, e.g., 192_168_16_2.
- **$SERVERHOST**. The host name of the MCP system.
- **$SERVERPORT**. The host's TCP port number from which the HTTP message was received.
- **$USERCODE**. The MCP usercode associated with the COMS dialog.
- **$WINDOW**. The name of the COMS Window associated with the WEBPCM Service.

You are responsible for constructing station names so that duplicates will be avoided.

Examples:

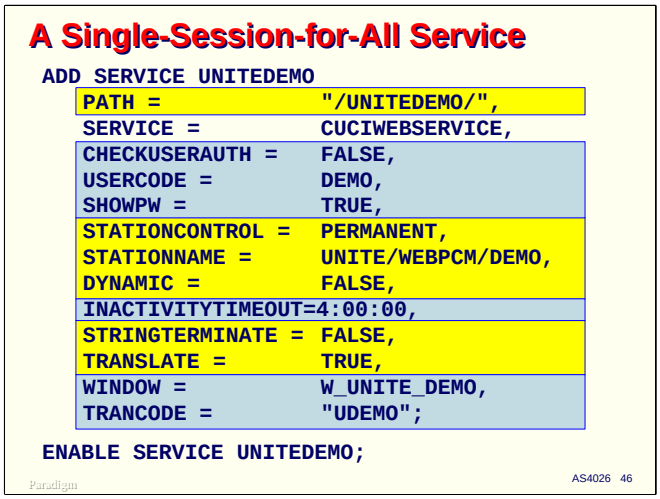
- `STATIONNAME = WEBPCM/PERM/STA`
- `STATIONNAME = $SERVERHOST/WEBPCM/$SERVERPORT/#`
- `STATIONNAME = $USERCODE/$REMOTEIPADDR/#`



Finally, there are two miscellaneous WEBPCM Service attributes:

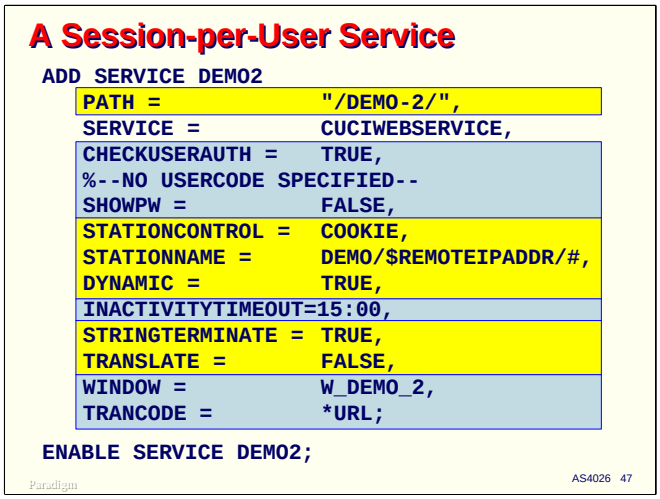
INACTIVITYTIMEOUT. This attribute specifies the maximum amount of time a station dialog will be held open with no activity going across it. It can be specified in terms of hours:minutes:seconds, minutes:sections, or just seconds. The default is 12 hours.

PROGRAMID. This attribute simply specifies a string that will be included in all CCF messages associated with the Service. It can be used to help identify which Service generated the message.



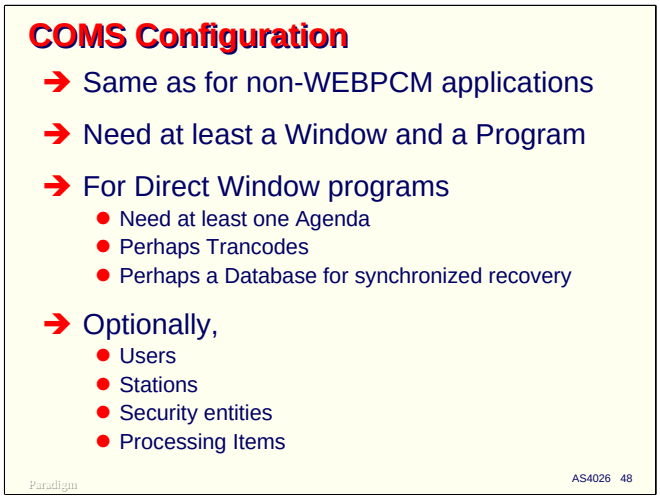
Now that we have looked at all of the WEBPCM Service attributes, let's go back and look at the sample Service definition we saw before. This example configures a Service that will funnel all web traffic from the Atlas Virtual Directory through a single COMS station and session to a specified Window and Trancode.

- The `PATH` setting identifies the corresponding Atlas Virtual Directory name. This is how Atlas links to a specific WEBPCM Service.
- The `SERVICE` attribute identifies the destination of the WEBPCM messages within CCF. It should always be `CUCUWEBSERVICE`.
- `CHECKUSERID=FALSE` indicates that the WEBPCM will not authenticate the user against the MCP USERDATAFILE. Authentication, if any, will be left to the application.
- The `USERCODE=DEMO` attribute specifies a constant MCP usercode that will be assigned to the COMS session. It is used in this case since the user is not being authenticated by WEBPCM.
- `SHOWPW=TRUE` indicates that if the application solicits credentials from the end user, it will be able to see the password in the value of the `$REMOTE-USER` pseudo-header value.
- `STATIONCONTROL=PERMANENT` indicates that this service will use a single COMS station for all users.
- `STATIONNAME` defines a constant name of `UNITE/WEBPCM/DEMO` for the COMS station.
- `DYNAMIC=FALSE` indicates that the single station will remain in the COMS CFILE. This allows the COMS administrator to specify non-default attributes and have them remembered.
- `INACTIVITYTIMEOUT=4:00:00` indicates that the station dialog should be closed after four hours of inactivity. The default agenda for the COMS Window will see a Window Close (Function Status = -50) message when this occurs.
- `STRINGTERMINATE=FALSE` specifies that string parameters to WEBAPPSUPPORT routines are to be padded out to their full length with spaces instead of terminated with a NULL.
- `TRANSLATE=TRUE` specifies that the WEBAPPSUPPORT routines will translate strings to the host character set. Since APPLICATIONCCS and CLIENTCCS are not specified, the default translation between ASCII and EBCDIC is used.
- `WINDOW=W_UNITE_DEMO` specifies the COMS window which will receive the messages.
- `TRANCODE="UDEMO"` specifies a constant trancode for all messages from this Service.

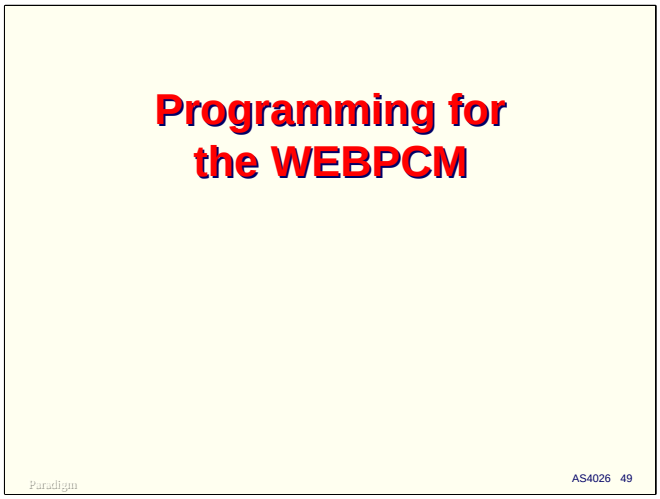


Unlike the prior example, which defined a single, permanent COMS station, with a constant COMS Trancode, and user authentication (if any) left up to the application, this example defines a WEBPCM Service which operates much more like traditional COMS stations.

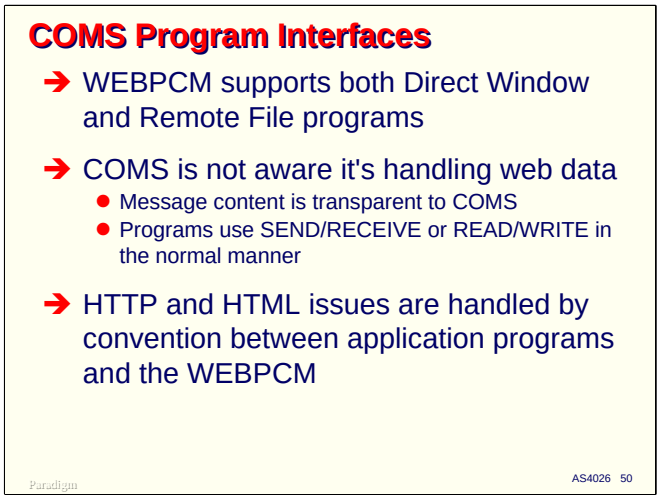
- Since `CHECKUSERAUTH=TRUE`, the WEBPCM will solicit a usercode and password from the end user when the station dialog is opened and validate them against the MCP USERDATAFILE.
- `SHOWPW=FALSE`, indicating that the application programs will not be able to see the user's password in the `$REMOTE-USER` pseudo-header.
- Session management will be controlled automatically by the WEBPCM using an HTTP cookie.
- Because this Service will potentially have multiple station dialogs open at once, the `STATIONNAME` attribute specifies a naming macro that will include the client's IP address and a unique number to resolve duplicates. If the "#" were left off of this macro, only one session would be accepted from the IP address at a time, since COMS supports only unique station names.
- `DYNAMIC=TRUE` indicates that the stations are temporary and will be removed from the COMS CFILE when their dialogs are closed. The inactivity timeout is 15 minutes.
- `STRINGTERMINATE=TRUE` specifies that the WEBAPPSUPPORT routines are to terminate string parameters with a NULL character instead of padding them out with spaces.
- `TRANSLATE=FALSE` specifies that no translation will be done. The application program will see the header and content values as the client sent them, most likely in ASCII.
- The `WINDOW` attribute specifies a different Window than the prior example, but it is possible for multiple WEBPCM Services to route messages to the same window.
- With `TRANCODE=*URL`, the second node of the URI path from the incoming HTTP request will be used as a COMS Trancode.



There is nothing special about the COMS CFILE configuration when using the WEBPCM. You simply need to make sure that any entities referenced by WEBPCM services, particularly Windows, Trancodes, and (if necessary) permanent Stations, are properly represented in the COMS configuration.



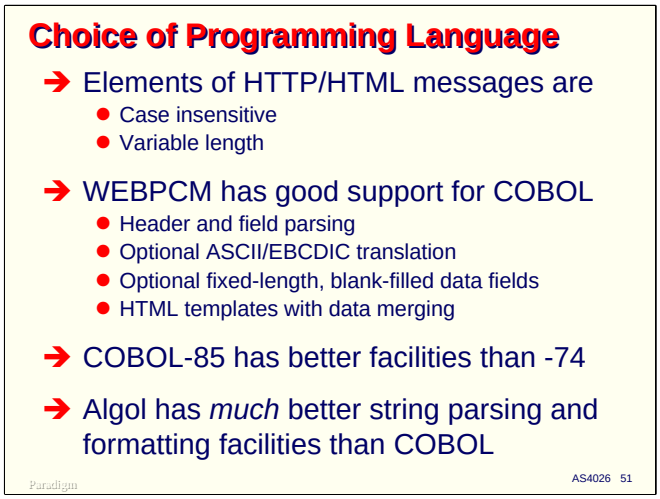
Now that we have seen how to configure Atlas and CCF for WEBPCM Services, we'll turn to the subject of writing COMS programs that interface with the WEBPCM.



The WEBPCM supports both Direct and Remote File window programs under COMS.

You program the interface to COMS exactly the same way as you do for other environments, and since the content of messages is transparent to COMS, COMS is not really aware that it is passing data intended for the WEBPCM. You define the COMS interface with Direct Window headers or remote files, and use Send/Receive or Read/Write statements as usual.

The part of programming that is different for WEBPCM interfaces is how you extract fields from the incoming messages and construct outgoing messages for the end user. You must follow the conventions established by the WEBPCM and use the WEBAPPSUPPORT library to manipulate both the incoming and outgoing message content.

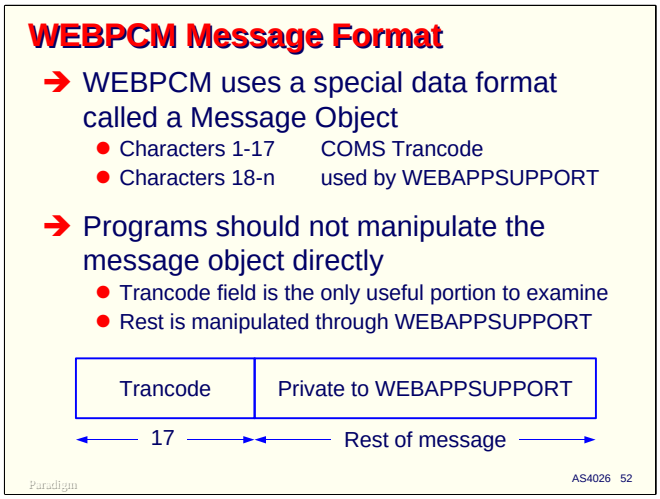


Programming directly for HTTP interfaces is generally much more challenging than it is for traditional green-screen terminal interfaces. Unlike messages from most green-screen interfaces, HTTP messages have a complex structure that must be parsed. The various elements are typically variable length, delimited by a variety of characters, and must be parsed in a case insensitive manner. It is not easy in any language, and is particularly difficult in COBOL.

Since COBOL-74 and -85 are the most commonly used languages for business applications under the MCP, they could use some help in this area, and the WEBPCM provides it. Routines in the WEBAPPSUPPORT library handled most of the details of parsing headers and URL-encoded data fields from HTTP messages, translating them to EBCDIC and presenting them in a form that is easy for COBOL programs to handle. HTML responses tend to be composed of lots of short, choppy sequences of variable-length text, which is relatively expensive to generate in COBOL. The WEBPCM provides a couple of ways to merge data values with an HTML template, which can dramatically ease this type of formatting.

If you have a choice between dialects of COBOL, COBOL-85 has generally superior facilities for the kinds of tasks you typically encounter in web-based programming. COBOL-74 is eminently usable with the WEBPCM, but not quite as convenient as COBOL-85.

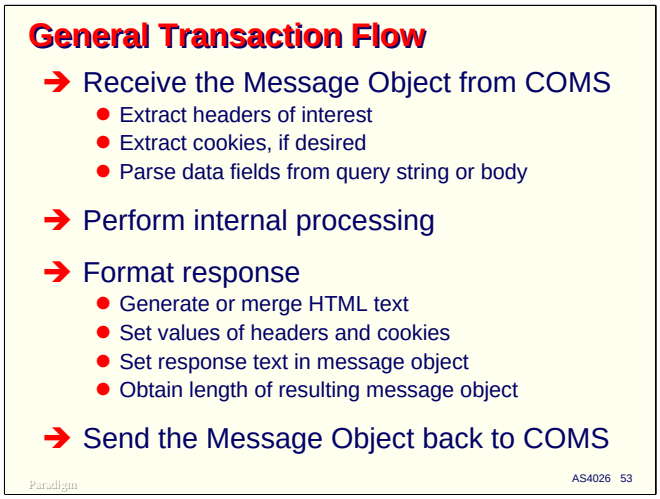
If you have Algol skills, you might want to consider using them for web-based programming. Algol is much, much more efficient than COBOL for this type of work, and is easier to use. It also supports the more efficient data representation modes provided by the WEBPCM, namely NULL-terminated strings and un-translated strings.



The most notable difference in the way you handle COMS messages originating from the WEBPCM is in the way you reference the actual data of the message. For most green-screen applications, your program sees what the terminal emulator sent. Your program knows the layout of the data, and simply extracts the fields from the message directly.

With the WEBPCM, your COMS program does not see the raw HTTP message. Instead, COMS passes it a special data structure built from the HTTP message by the WEBPCM. This data structure is called a **Message Object**.

The first 17 characters of the message object are the COMS Trancode as determined by the WEBPCM Service. This is the only part of the message your program should normally examine directly. The rest of the message is encoded in a format known only to the WEBPCM, and likely to change from software release to software release. Instead of accessing data in this private area directly, you make calls on the routines in the WEBAPPSUPPORT library to do this for you. Much of the work in learning to program for the WEBPCM is in learning the routines of the WEBAPPSUPPORT library and what they do.



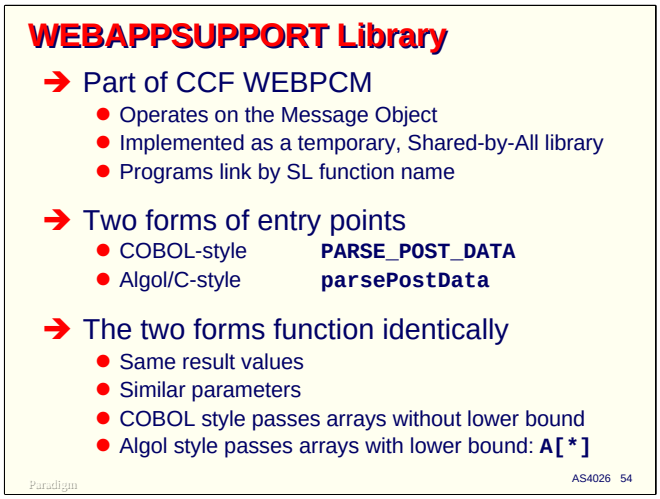
Except for the WEBAPPSUPPORT library calls, the general program flow for processing messages originating from the WEBPCM is very similar to that for other types of COMS programming.

First, you receive a new message from COMS. You program will probably want to extract some headers from the message object to identify the message and its source. If you are using cookies, WEBAPPSUPPORT provides a means to access those as well. In most cases, you will also want to parse data fields from the query string or content area of the message.

Once the input message is parsed and the data edited according to your application's business rules, there is typically some internal processing that must be done, often involving data base queries and updates.

As a product of the internal processing, you will typically need for format some sort of a response. For the WEBPCM, this response will usually take the form of HTML text. You can generate the HTML text yourself, or use one of the template merging facilities provided by WEBAPPSUPPORT, or a combination of techniques. Once the text is prepared, you place it in the message object using a `SET_CONTENT` call on WEBAPPSUPPORT. You may also need to set the value of certain HTTP headers and cookies, which you also do through the WEBAPPSUPPORT routines `SET_HEADER` and `SET_COOKIE`.

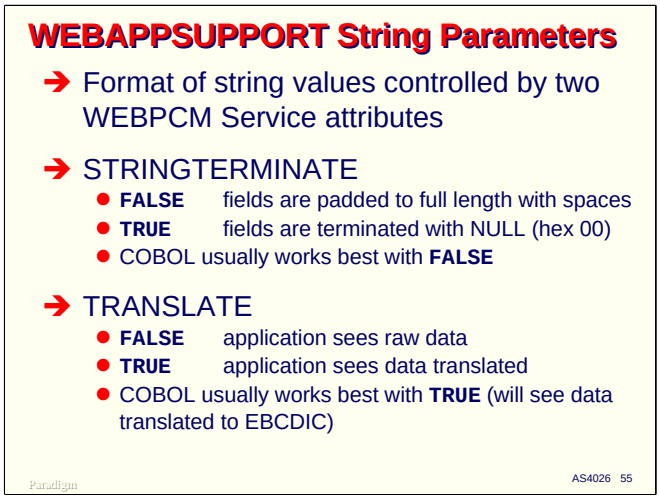
The effective length of the message object changes as you make calls on WEBAPPSUPPORT routines, so before you send the response back to the WEBPCM, you must make a call on the WEBAPPSUPPORT routine `GET_MESSAGE_LENGTH` to determine the object's current length. You then use this length in a COMS Send or Write statement. If your response is too long to fit in the data area your program has allocated to the message object, you can break the reply into pieces and send them one at a time.



The WEBAPPSUPPORT library is a program's primary tool for accessing HTTP messages and manipulating the WEBPCM message object data structure. It is a temporary, shared-by-all system library. Programs link to it using its SL name, `WEBAPPSUPPORT`.

The library has a number of routines that can be called by COMS programs. Each of these routines have two entry points in the library, one oriented primarily to COBOL programs, and one oriented primarily to the other languages, especially Algol. The COBOL-style names are in upper case, with embedded underscores. The Algol-style names are in mixed case without underscores.

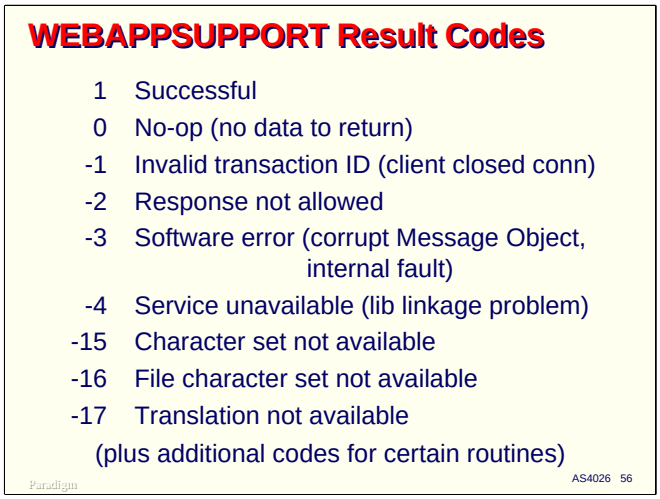
Both entry points for each routine perform the same function and have the same list of parameters. The primary difference between them is that the COBOL-style routines pass string (EBCDIC array) parameters without lower bounds, while the Algol-style routines pass them with lower bounds (i.e., using the Algol notation `A[*]`). The `SET_CONTENT` routine uses a 1-relative index into a user's buffer area; `setContent` uses a 0-relative index.



As we discussed earlier, the data representation in string parameters for WEBAPPSUPPORT routines is controlled by two attributes for each WEBPCM Service.

If `STRINGTERMINATE` is False, string values are padded with spaces to their full declared length. If True, the values are terminated with a NULL (hex 00) character. The first setting is usually more convenient for COBOL programs to process, while the second is more efficient and is easily handled by Algol programs.

If `TRANSLATE` is False, string values are delivered to the program in the same character coding as used by the client web browser that submitted the request. If this attribute is True, the string values are translated to the host's character set configured for the Service, which by default is EBCDIC. The first setting is slightly more efficient and can be handled easily by Algol programs, while the second is more convenient for COBOL.

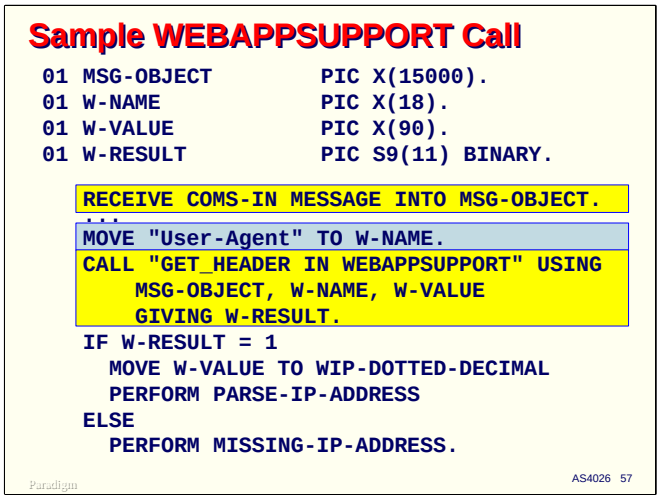


All of the WEBAPPSUPPORT routines are called as functions (typed procedures) and return and integer result code. Calling programs should check this value to be sure the routine completed successfully.

A return value of 1 indicates that the routine completed successfully.

A return value of zero indicates that the routine was not able to fulfill the caller's request, but this does not necessarily imply that there was an error. In many cases, it simply means that the header or other data item the caller was trying to access did not exist in the message.

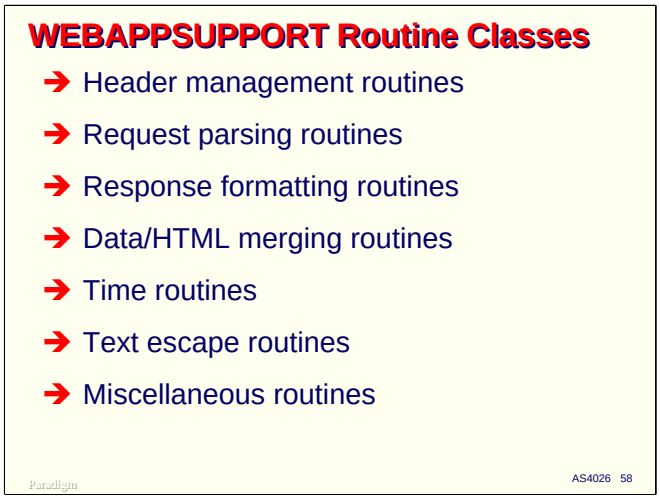
Negative return values indicate some sort of error. In addition to the negative codes shown on this slide, there are some additional ones that are specific to certain routines, especially those that merge data values with an HTML template.



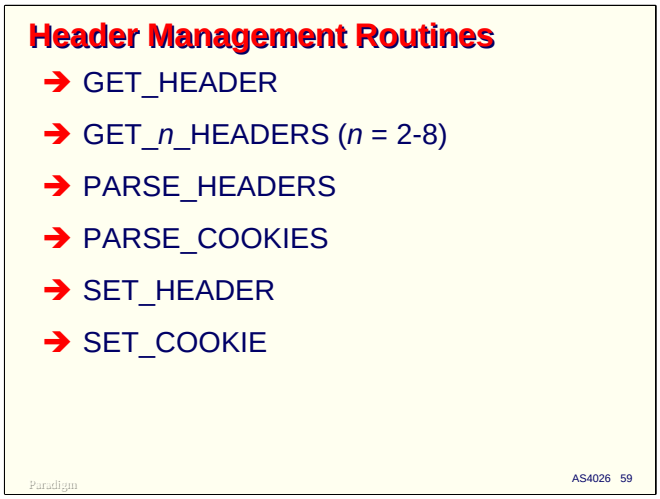
To see what a typical WEBAPPSUPPORT call looks like, this slide shows an example in COBOL that fetches a header value from the message object.

The GET_HEADER routine will search the HTTP message for a specified header, and if found, return it's value to the caller. For the purpose of this example, assume the WEBPCM Service has been declared with STRINGTERMINATE=FALSE and TRANSLATE=TRUE.

- The first parameter contains the message object the program received from COMS.
- The second parameter is a string containing the name of the header (in this case, it is the User-Agent header, which is intended to identify the browser which submitted the message).
- The third parameter is the area where the header's value will be placed, if found in the HTTP message. Since STRINGTERMINATE=FALSE, this area will be padded with spaces to its full length of 90 characters when the header value is stored there.
- The GIVING clause on the CALL statement specifies where the result code will be stored.



The WEBAPPSUPPORT routines fall into several general classes. We'll discuss them by class in the following slides.



There are a number of routines that will access and manipulate HTTP headers in the message object.

GET_HEADER retrieves a header from the message object, as shown in the previous example.

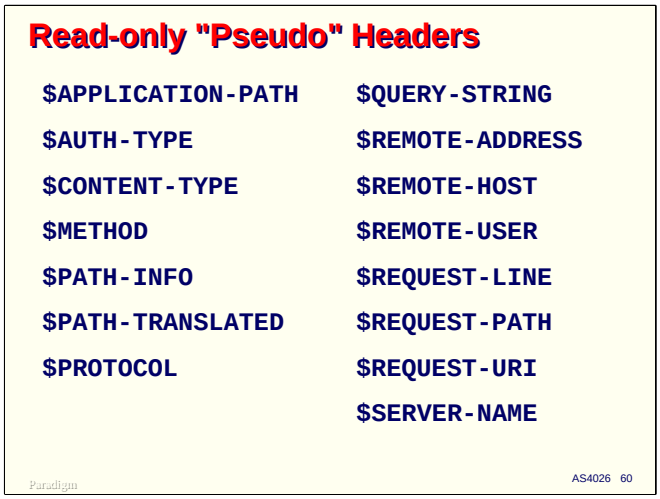
GET_*n*_HEADERS (where *n* = 2-8) works just like GET_HEADER, but allows you to retrieve multiple headers in one call.

PARSE_HEADERS is useful when you want to examine the headers in an HTTP request, but you don't know what the names of the headers are. This routine extracts a specified number of headers (or the number present, if fewer) from the message object and builds a table of name/value pairs with fixed-length columns. The calling program can then traverse the table to see which headers are present.

PARSE_COOKIES is similar in concept to PARSE_HEADERS, but is oriented to a special type of HTTP header, the "cookie." Cookie values have several parts. This routine parses the values from the incoming HTTP request into their parts, then builds a table with one row per cookie and separate fixed-length fields for each part of the value.

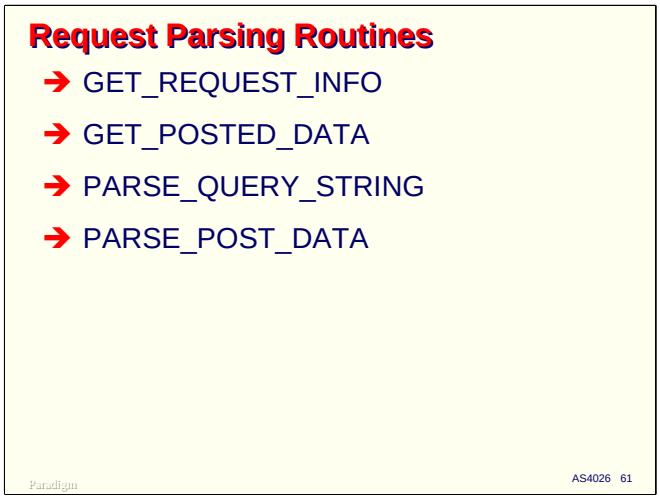
SET_HEADER does the opposite of GET_HEADER. It takes a header name and value and formats it in the HTTP response portion of the message object.

SET_COOKIE is similar to SET_HEADER, but stores a cookie header in the response portion of the message object.



In addition to the header values actually present in an HTTP request, the WEBPCM derives the value of a number of so-called "pseudo-headers" from the request and the server environment. These can be retrieved using the GET_HEADER and GET_*n*_HEADERS routines, but not altered with SET_HEADER.

- **$APPLICATION-PATH** returns the first two nodes of the URI path, i.e., the Virtual Directory node and the node following that. Any URL encoding is decoded.
- **$AUTH-TYPE** returns the authorization scheme for the request or a result code of zero if there is no authorization header in the request.
- **$CONTENT-TYPE** returns the MIME type for the request, e.g., "text/html".
- **$METHOD** returns the request method, usually "GET" or "POST".
- **$PATH-INFO** returns the rest of the URI path after the first two nodes but before the query string. Any URL encoding is decoded.
- **$PATH-TRANSLATED** returns the same thing as $PATH-INFO, but translated to file pathname format. Not to be used if COMS synchronized recovery is in effect.
- **$PROTOCOL** returns the level of HTTP protocol from the request line.
- **$QUERY-STRING** returns the text of the query string from the request line.
- **$REMOTE-ADDRESS** returns the IP address of the client in dotted-decimal format.
- **$REMOTE-HOST** returns the fully qualified DNS name of the client.
- **$REMOTE-USER** returns the name of the remote user from the authorization header in the request. If no authorization header is present, returns a null string. If SHOWPW=TRUE for the Service, the string will include the password, delimited from the user name by a colon.
- **$REQUEST-LINE** returns the complete request line from the HTTP request.
- **$REQUEST-PATH** returns the complete URI path name. Any URL encoding is decoded.
- **$REQUEST-URI** returns the complete URI path, including any query string.
- **$SERVER-NAME** returns the host name of the server.



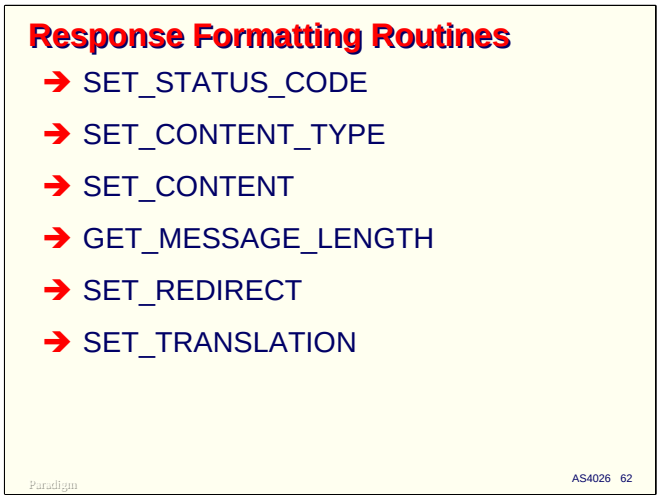
There are four routines that will extract information about and values from HTTP request in the message object.

GET_REQUEST_INFO returns several integer values that indicate the lengths of various portions of the request, e.g, length of the request line, length of the query string, length of the content (body) of the message, and total length of the HTTP message.

GET_POSTED_DATA simply returns raw bytes from the content (body) of the HTTP request. This routine may be called multiple times if the content size is too large for the program's receiving area.

PARSE_QUERY_STRING is similar in concept to the PARSE_HEADERS routine. It parses the entire query string and returns a table of name/value pairs in fixed-length columns, with any URL encoding decoded to plain text.

PARSE_POST_DATA does the same thing as PARSE_QUERY_STRING, but parses the message content (body) instead of the query string, also decoding any URL encoding.



The WEBAPPSUPPORT library provides several routines to set up the response are in the message object to send a reply to the end user.

SET_STATUS_CODE sets the HTTP status code and textual description in the response area of the message object. If you do not call this routine, a default status of "200 OK" is provided automatically by the WEBPCM. See the HTTP specification for the full list of status codes. In addition to the default 200 response, the following are typically the most useful for application program responses:

- `400` Bad request.
- `401` Unauthorized. Sending this will cause the browser to request credentials from the user. The next input from the browser should contain an Authentication header with the username and password supplied by the end user. You can obtain these values from the `$REMOTE-USER` pseudo-header.
- `403` Forbidden operation.
- `404` Not found.
- `501` Not implemented.

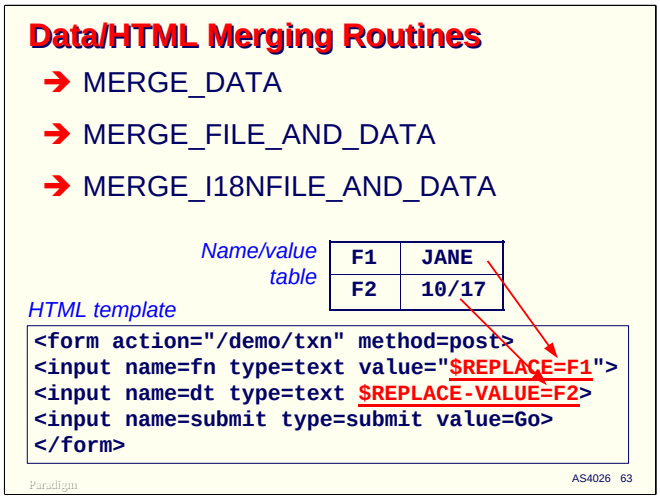
SET_CONTENT_TYPE sets the MIME type in the message object that describes the format of the response data. If you do not call this routine, the WEBPCM sets a default type of "`text/html`" if the response contains a body.

SET_CONTENT stores bytes of data in the content (body) area of the message object response.

GET_MESSAGE_LENGTH returns the current length of the message object. The effective length of this data area changes as calls are made to set response headers and content. Before sending the response to COMS, you should call this routine to obtain the current effective length that should be passed in the COMS Send or Write statement.

SET_REDIRECT sets up the message object to return a "redirect" result the the originating browser. A redirect response is used when the server does not generate the response itself, but knows the URL where the response can be found. When the browser receives this response, it automatically generates another request to the URL contained in the response. This type of response is often used when the request must be processed by another server, or when the request can be satisfied by some static content for which the URL is known.

SET_TRANSLATION can be used to change the Code Character Sets (CCS) for the host and client systems at run time. A call on this routine overrides the `APPLICATIONCCS` and `CLIENTCCS` attributes defined in the CCF PARAMS file, and remains in effect until changed by another call or the WEBPCM is restarted.



WEBAPPSUPPORT provides two methods to merge data values with an HTML template. You prepare a template for the HTML response and mark the places in the template where actual values will be substituted at run time. You then build in your program a table of name/value pairs for the data that is to be plugged into the template. The names in this table match the markers coded in the template, and the merge routines simply copy the template to a result buffer, replacing each marker with its corresponding value from the table. The result buffer can then be set into the message object response area.

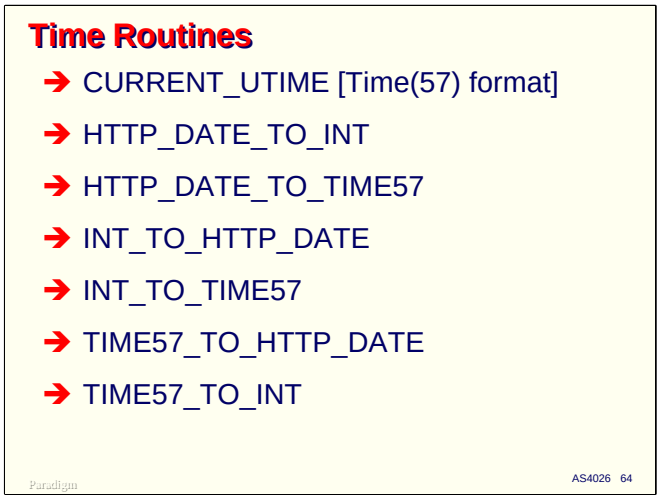
In addition to one-for-one replacement, the merge routines permit looping through repeated names, which is useful for populating HTML table structures.

This facility works much like parameter substitution in a CANDE `DO` file, and can be an especially useful and efficient alternative to performing tedious string processing in COBOL.

`MERGE_DATA` accepts a table of name/value pairs in a memory buffer, an HTML template in another memory buffer, several parameters that describe the buffers and how the data values are to be treated, and stores the merged HTML text in a result buffer.

`MERGE_FILE_AND_DATA` is similar to `MERGE_DATA`, but reads the template from a disk file rather than a memory buffer. The merged result is left in a memory buffer.

`MERGE_I18NFILE_AND_DATA` works the same as `MERGE_FILE_AND_DATA`, but allows the character set of the template file to be overridden by the value of an additional parameter. It is normally used only for files containing non-Latin character sets.



Some HTTP headers contain time values, which are always represented in one of three standard text formats, and are based on the Universal (Greenwich) time zone. WEBAPPSUPPORT contains several routines that can translate the HTTP formats to more useful forms.

CURRENT_UTIME returns the server's current time of day in Algol(57) format. This is the same format as Time(7), but represented as Universal Time. Time(7) and Time(57) are real (single-precision floating point) values and have the following bit fields:

- [47:12] Year (1900-2035)
- [35:06] Month (1-12)
- [29:06] Day (1-31)
- [23:06] Hour (0-23)
- [17:06] Minute (0-59)
- [11:06] Second (0-59)
- [05:06] Day of the week (0 = Sunday, 1 = Monday, … 6=Saturday)

HTTP_DATE_TO_INT converts a string value from one of the three HTTP standard time formats to a integer representing the number of seconds since year 0, day 0.

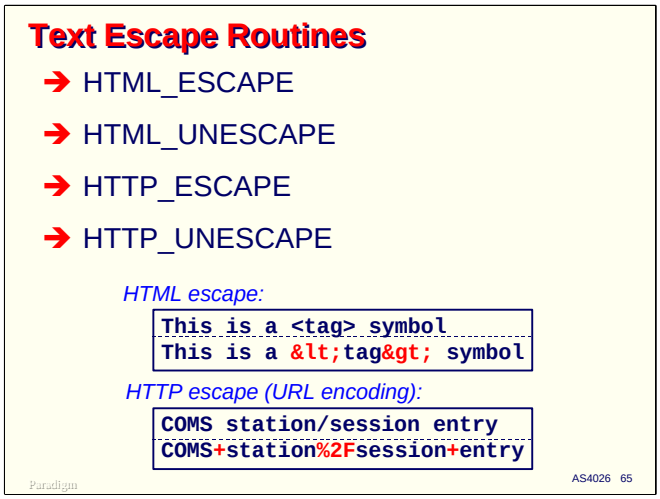
HTTP_DATE_TO_TIME57 converts a string value from one of the three HTTP standard time formats to real value in Time(57) format.

INT_TO_HTTP_DATE converts an integer value from the seconds-since-year-0 format to the RFC 1123 date format.

INT_TO_TIME57 converts an integer value from the seconds-since-year-0 format to a real value in Time(57) format.

TIME57_TO_HTTP_DATE converts a real value from Time(57) format to a string value in RFC 1123 format.

TIME57_TO_INT converts a real value from Time(57) format to an integer in seconds-since-year-0 format.



HTML allows any sequence of text to be represented, but uses a few special characters (such as "<", ">", "&" and the double quote) as delimiters for tags and enumerated character values. In addition, some characters in the extended ASCII character set cannot be represented directly. If you need to format text values in HTML which contain these special characters, you must "escape" (encode) these values to represent the special characters differently. WEBAPPSUPPORT supplies routines to encode and decode HTML strings.

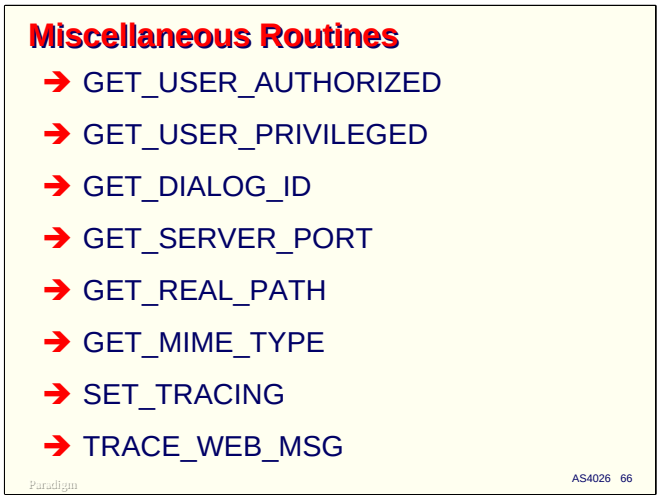
HTML_ESCAPE encodes a string of plain text so that it will be transparent to an HTML parser and will be displayed properly when sent to a browser.

HTML_UNESCAPE decodes a string of HTML-encoded text back to plain text.

Similarly, the nodes and query string elements of URI paths must be encoded if they contain certain special characters, including "/", "?", "+", ":", "&", and "%". This type of encoded is called "URL encoding" and is supported by two WEBAPPSUPPORT routines.

HTTP_ESCAPE URL-encodes a string of text so it will be transparent in a URL or URI path.

HTTP_UNESCAPE decodes URL-encoded text to translate the escaped characters back to plain text.



There are a few miscellaneous routines that do not fall into any other convenient category.

GET_USER_AUTHORIZED determines whether a user has been authenticated against the MCP USERDATAFILE.

GET_USER_PRIVILEGED determines whether an authenticated user is also a privileged user.

GET_DIALOG_ID returns the WEBPCM station dialog identifier for the current dialog. This value must be used by the application to format HTML hidden fields and links when `STATIONCONTROL=HTML`.

GET_SERVER_PORT returns the TCP port number from which Atlas received the HTTP request.

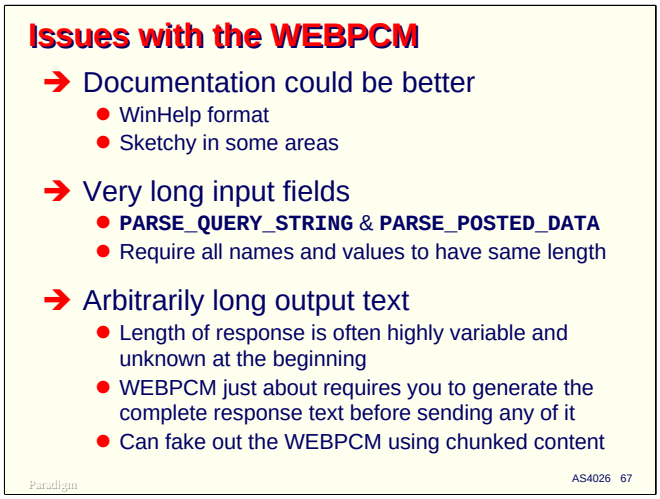
GET_REAL_PATH translates a URI path name to a physical disk file PATHNAME attribute. This routine should not be used when DMSII synchronized recovery is in effect for the application.

GET_MIME_TYPE returns the MIME type for a specified path name. This routine should not be used when DMSII synchronized recovery is in effect for the application.

Finally, there are two routines that can be used to control diagnostic tracing capabilities in the WEBAPPSUPPORT library. You may find these useful when debugging WEBPCM applications.

SET_TRACING turns tracing of WEBAPPSUPPORT calls on or off for the calling program. The trace information is written to the WEBAPPSUPPORT trace file.

TRACE_WEB_MSG writes a text string to the WEBAPPSUPPORT trace file.

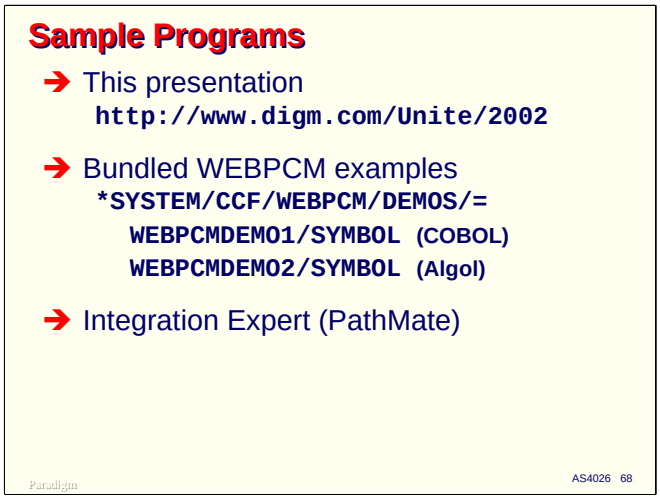


The WEBPCM is a great facility, but like all other facilities, there are a few items which could use improvement.

Foremost of these is the documentation. The WEBPCM is documented in the CCF Administration and Programming Guide. Help files are great, and have their place, but as the sole source of documentation for tool that requires study and integration of various parts, neither the WEBPCM nor the rest of CCF are one of those places. In addition to the medium itself, the document as currently structured requires you to click on numerous railroad track elements to learn the syntax and semantics of individual items. This is extremely tedious, and makes it difficult to avoid Second Order Ignorance (not knowing what you don't know). The discussion of defaults and the interaction between syntactic elements is often sketchy or non-existent. All of this makes CCF and the WEBPCM somewhat difficult to learn initially. The entire CCF documentation would be much better if done in the form of a traditional reference manual and distributed as PDF.

The way that `PARSE_QUERY_STRING` and `PARSE_POST_DATA` work is convenient for most typical cases, but almost impossible to use in those cases where there are very large numbers of fields, or where some of the field values are very long. The values are returned in a table, which can have only a fixed maximum size, and the columns within the table must all be of the same fixed width. If a value exceeds the width allocated for it, these routines simply stop parsing at that point and return an error. There is no way to extract a single field from either the query string or message content aside from using `GET_POSTED_DATA` and parsing the URL-encoded fields yourself, which is very difficult in COBOL. The WEBPCM could use a routine like `GET_HEADER` that would operate against the query string or message content and retrieve a single, named field. It also needs the ability for a program to obtain a directory of all fields in a query string or content area, regardless of the limitations on a program's internal parameter sizes (e.g., 65K bytes for COBOL-74 01-records).

Finally, the WEBPCM does not make is very easy to generate responses of arbitrary length. Very long responses need to be broken up into segments, which is fine, but the WEBPCM requires you to set a `Content-Length` header for the total length in the response before sending the first segment. This approach just about requires that you generate the complete response before you send any of it, which in comes cases could be hundreds of thousands of bytes. You can fake out the WEBPCM by setting a very large value for `Content-Length` and then using HTTP chunked responses formatting, but this is, frankly, a kludge. The WEBPCM should support chunked responses directly.

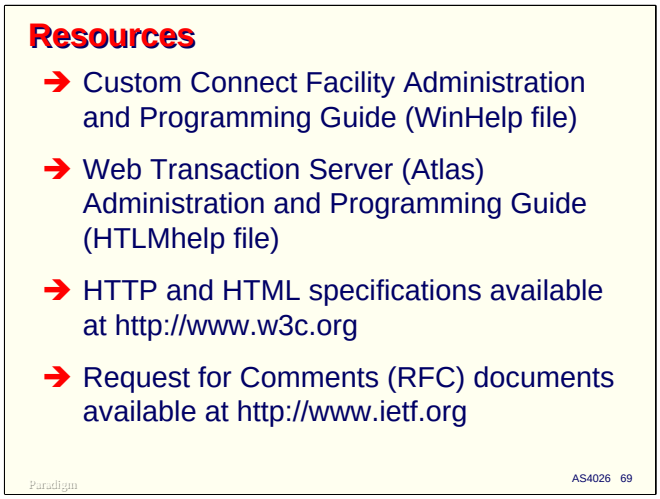


This is not time in this presentation to cover all of the issues with web programming or the WEBAPPSUPPORT library in any detail. The best way to learn more about programming for the WEBPCM is to look at some sample programs.

This presentation includes a sample COBOL program that uses the WEBPCM, which we will see in just a minute. You can get a copy of the source code for this program from our web site at the URL shown on the slide.

Unisys delivers two WEBPCM sample programs with the system release, one in Algol and one in COBOL-74. The source code is included under the `*SYSTEM/CCF/WEBPCM/=` or `*SYSTEM/COMS/CCF/WEBPCM/=` directories, depending on the release level you are running. These samples illustrate use of essentially all of the WEBAPPSUPPORT routines. You can run these sample programs from the `ATLASSUPPORT` administrative web site on your system. The necessary CCF and COMS configuration settings should have been installed for you by Simple Install or the Install Center utilities.

As always, the Integration Expert, better known as PathMate, has a complete example of using the WEBPCM, including a good illustration of HTML template merging.



Configuration of the WEBPCM and use of the WEBAPPSUPPORT library is documented with the rest of the CCF facilities in the Custom Connect Facility Administration and Programming Guide.

Configuration of the Atlas web server is covered in the Web Transaction Server Administration and Programming Guide.

If you are interested in learning more about HTTP and HTML, the specification documents are available in a variety of formats from the World Wide Web Consortium's web site at http://www.w3c.org. Most bookstores also carry a daunting variety of books on HTML and web server programming.

Many documentation sources, especially the W3C specifications, refer to Request for Comment (RFC) documents. You can obtain these online from the Internet Engineering Task Force's web site at http://www.ietf.org.

End

Using the WEBPCM

Session AS4026

2002 UNITE Conference