Creating Dynamic Web Content with CA-Visual Objects 2.5

By Thomas W. Elledge

CA-World 2000
eBusiness Solutions in Internet Time
JP115SN

Introduction

The World Wide Web (WWW) has become increasingly important in the delivery of information in today's technology literate world. With the staggering growth of the Internet as seen in Figure 1, businesses and government agencies have awakened to the potential of the Internet for information delivery and increased visibility. With large amounts of information stored in internal databases within these organizations, the potential for electronic delivery of this information to the customer is enormous.

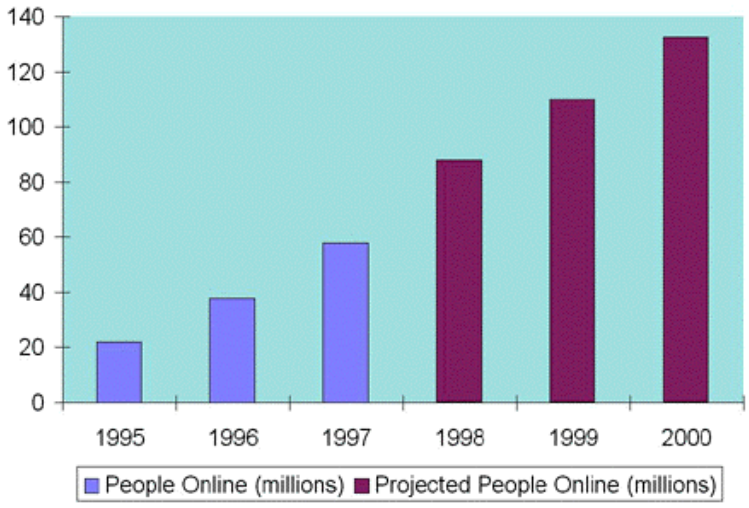

Figure 1: Internet Population Projected Growth (Source CommerceNet)

This session will focus on techniques that allow developers to utilize CA-Visual Objects 2.5 to create and deliver dynamic Web content. We will examine several technologies that are built into the product for this purpose. We will explore the use of Common Gateway Interface (CGI), Internet Server Application Programming Interface (ISAPI) and Active Server Page (ASP) classes to demonstrate how these technologies are implemented in CA-Visual Objects 2.5.

By combining your knowledge of CA-Visual Objects along with some Internet programming knowledge, interactive Internet applications can be created to provide a full range of applications. We will begin by looking at the basics of publishing on the Web and go into detail on how we can exploit the power of CA-Visual Objects 2.5 to create dynamic Web content that will be displayed in an Internet browser.

Web Servers

The first component in serving CA-Visual Objects 2.5 information to the Internet is a Web server. Web servers are often referred to by the primary protocol used on the Web – HTTP (HyperText Transport Protocol) servers. An ever growing number of Web servers are available running on a variety of operating systems. UNIX-based servers remain the most prevalent on the Web today with MS-Windows NT coming on strong. Since the focus of this paper is CA-Visual Objects 2.5, we will restrict our discussion to servers that run on the MS-Windows platform.

The Web server that we have standardized on for our Internet work is the Microsoft Internet Information Server (IIS), which is available as a service in all of the Windows 2000 products. IIS provides powerful and scalable Web services with tight integration into the operating system and a very complete compliment of management tools.

HyperText Markup Language

HyperText Markup Language (HTML) is the heart and soul of the Web. It provides a platform-independent specification for the creation of hypertext documents containing forms, clickable image maps, graphics, hypertext links and much more. HTML is comprised of formatting commands, referred to as tags, which are used to designate text as headings, paragraphs, lists, quotations, emphasized and various other components. Tags also exist for including images within the document, creating online forms that accept user input for back-end processing and for creating hypertext links that allow the document being viewed to access other documents on the Internet. Hypertext links can also trigger file downloads, access anonymous FTP sites, search database engines, display video clips and play sound files.

One of the key strengths of HTML is that a document conforming to the HTML standard can be viewed on any computer for which a Web browser exists without regard to the native operating system running on the computer. For example, the same document can be viewed by a user running Netscape in MS-Windows, someone using Lynx on a UNIX system, or even a blind person using special software.

HTML Forms

Before we launch into a discussion of how to create dynamic content, we need to discuss how a CA-Visual Objects 2.5 back-end application obtains information from the client browser via the Web server. One of the components of the HTML specification is an HTML Form. An HTML Form allows a user to enter information that is passed to the back-end application, which is usually located on the Web server. This Common Gateway Interface (CGI) compliant application processes the information and returns an appropriate response to the user. Consider for a moment the Form depicted in Figure 2. This Form requests mailing information (both electronic and postal) from the user for the purpose of obtaining a brochure outlining the services provided by COMPCON Software.

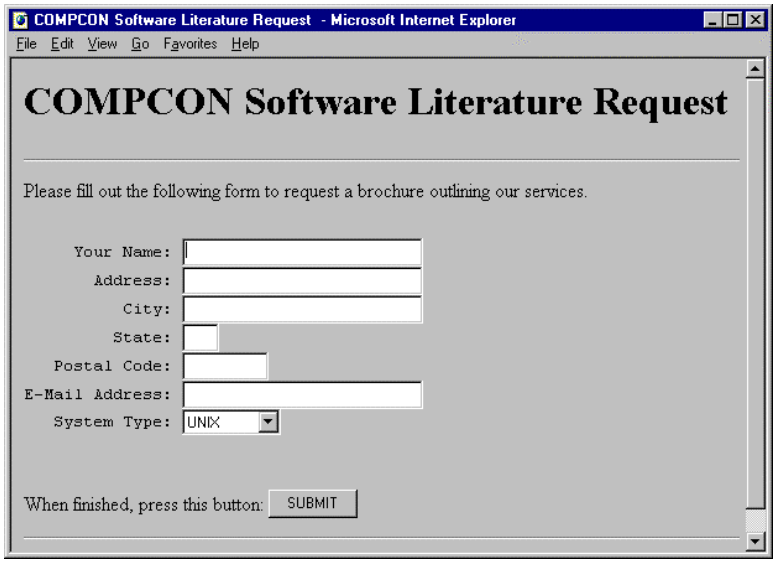

Figure 2: Literature Request Form

When the user has completed the Form, they press the SUBMIT button. The information from the Form is then submitted to the CGI application via the Web server.

The Form in Figure 2 was produced by the following HTML code:

```
<HTML>
<HEAD>
<TITLE> COMPCON Software Literature Request </TITLE>
</HEAD>

<BODY>
<H1> COMPCON Software Literature Request </H1>

<HR>
Please fill out the following form to request a brochure outlining our services.
```

```
<FORM ACTION="www.compconsoftware.com/scripts/brochure.exe" METHOD="POST">

<PRE>
     Your Name: <INPUT TYPE="text" SIZE=35 NAME="name">
       Address: <INPUT TYPE="text" SIZE=35 NAME="address">
          City: <INPUT TYPE="text" SIZE=35 NAME="city">
         State: <INPUT TYPE="text" SIZE=2  NAME="state">
   Postal Code: <INPUT TYPE="text" SIZE=10 NAME="postal">
E-Mail Address: <INPUT TYPE="text" SIZE=35 NAME="email">
   System Type: <SELECT NAME="system">
                    <OPTION SELECTED>UNIX
                    <OPTION>Windows 3.x
                    <OPTION>Windows 95
                    <OPTION>Windows NT
                    <OPTION>OS/2
                    <OPTION>Macintosh
                    <OPTION>Other
                    </SELECT>

</PRE>

When finished, press this button: <INPUT TYPE="submit" VALUE="SUBMIT">

<HR>
</FORM>
</BODY>
</HTML>
```

The FORM Tag

The Form section of an HTML document is delineated by a starting <FORM> tag and terminated with an ending </FORM> tag. Any HTML code may be placed within these tags with the exception of another Form tag. The Form in Figure 2 is initiated by the following statement:

```
<FORM ACTION="www.compconsoftware.com/scripts/brochure.exe" METHOD="POST">
```

The Form attributes included in this tag are:

ACTION

The action to be performed when the user presses the SUBMIT button. This is normally the name of the back-end CGI application.

METHOD

The method used to transfer the user-supplied information to the back-end CGI application. The most commonly used methods are GET and POST.

In our example, the Form will pass its data via the POST method to an application named *brochure.exe*, which lives in the *scripts* directory of the Web server located at www.compconsoftware.com. Brochure.exe is our back-end CGI application.

Form Methods

Before we proceed, we need to discuss Form methods. There are two HTML Form methods which are commonly used. The method used determines the way in which the Web server passes the Form data to the back-end CGI application. The standard methods are:

- **The GET Method** – If the HTML Form tag contains METHOD="GET", the CGI application will receive the encoded Form input in the environment variable QUERY_STRING.
- **The POST Method** – If the HTML Form tag contains METHOD="POST", the CGI application will receive the encoded Form input via standard input handle (StdIn). The Web server will not send an EOF at the end of the data stream. The CGI application must use the environment variable CONTENT_LENGTH to determine the number of bytes that should be read from StdIn.

How this data is accessed and what happens to it is up to the back-end application. There are three primary classes included in CA-Visual Objects 2.5 that allow Form data to be processed and some type of information returned to the client browser. The **IScriptingContext** class is used to produce ASP components. In addition, there are the **HTTPCgiContext** class, which corresponds to the Common Gateway Interface (CGI) specification, and the **HTTPExtensionContext** class, which allows interaction according to the Microsoft Internet Server API (ISAPI) specification. We will now turn our attention to the CGI and ISAPI specifications.

Common Gateway Interface

The Common Gateway Interface (CGI) is an Internet standard for interfacing external gateway applications with HTTP servers. It provides a very flexible mechanism for processing data sent from a client browser via the Web server to a back-end CGI compliant application.

When a user presses the SUBMIT button on an HTML Form, all of the information entered onto the Form is sent by the client browser via HTTP to the Web server. The server packages the browser response, launches the CGI application and passes the response to it. Once started, the CGI application retrieves the details of the response package created by the Web server, processes the request, and generates a response for the server to send back to the browser. The response can be anything that the developer deems necessary. It can be as simple as an "OK" to a complete database report including graphics. The CGI application returns the response and then terminates. When the Web server senses that the CGI application has terminated, it sends the response to the browser. From the user perspective, the process is transparent and appears the same as fetching a static document.

To construct a CGI application, the developer needs to know three basic techniques:

- **Data In** – How to retrieve the NAME/VALUE pairs created by the server
- **Parameter Access** – How to read the CGI variables
- **Data Out** – How to send the results back to the server

The CGI Process

In a nutshell, a CGI-complaint application receives encoded information from the Web server via *standard in* (StdIn) and returns a result to the HTTP server via *standard out* (StdOut). In addition, various other parameters are available to the back-end application as environment variables. This process reflects the fact that CGI was originally designed for Web servers running on the UNIX platform.

Data In and Parameter Access

When the Web server invokes our CA-Visual Objects 2.5 CGI application, we must provide some mechanism by which to retrieve the HTML Form variables from the Web server. Before the advent of the HTTPCgiContext class, our application was required to open the standard input handle (StdIn) to retrieve the Form variables and the standard output handle (StdOut) in order to return our response to the Web server. To access these handles, we utilized the AllocConsole() Win 32 API call. Once the application opened StdIn, we had to resort to the FRead() function to do a low-level read of the data using the contents of the CONTENT_LENGTH environment variable. The Form data retrieved via FRead() from the StdIn handle was in the format of a stream of name/value pairs separated by an "&" character. The name/value pairs were then broken up and placed into a two-dimensional array. But we still weren't finished, as each of these values are encoded in a standard URL format where spaces were changed to "+" and special characters were encoded into hexadecimal (%xx). This string had to be decoded by the CGI application.

But all of that is ancient history with the **HTTPCgiContext** class. All of these details are handled for us by the **ReadClient**() method. **ReadClient**() reads information from the body of the Web client's HTTP request. It can be used to read data from an HTML Form that uses the POST method. But there is an even easier way to access the Form data in CA-Visual Objects 2.5. The **Init**() method of the **HTTPCgiContext** class uses **ReadClient**() internally to read data for a POST request and stores them into the property, aArgs, which can be retrieved by the **GetParamValue**() method.

In our application, accessing HTML Form data is as easy as:

```
FUNCTION GreetEm

       LOCAL oCGI   AS HTTPCGIContext
       LOCAL cHtml  AS STRING
       LOCAL sFName AS STRING
       LOCAL sLName AS STRING

       oCGI := HTTPCGIContext{}

       sFName := oCGI:GetParamValue("First")
       sLName := oCGI:GetParamValue("Last")

       cHtml := oCGI:HTTPMsg("Welcome to our world, " + sFName + " " + sLName)

       oCGI:Write(cHtml)
       oCGI:Close()

RETURN
```

In this function, we create an instance of the **HTTPCgiContext** class and access the HTML Form variables with calls to the **GetParamValue**() method. Once we have obtained the values, we can process them as we would any character variable.

Server Variables

In addition to the encoded Form input, the Web server passes data about the request and itself as server variables. Many of these variables are useful to the CGI application and may be retrieved via calls to the CA-Visual Objects 2.5 **GetServerVariable**() method of the **HTTPContextAbstract** class. The server variables that are commonly available are depicted in the table below:

AUTH_TYPE	The authentication method that the Web server uses to validate users when they attempt to access a protected script. For example, if Basic authentication is used, the string will be "Basic." For NT Challenge-response, it will be "NTLM". If the string is empty, then no authentication is used.
CONTENT_LENGTH	The number of bytes which the script can expect to receive from the client.
CONTENT_TYPE	The data type of the content. Used with queries that have attached information, such as the HTTP queries POST and PUT.
GATEWAY_INTERFACE	The revision of the CGI specification used by the Web server. The current version is CGI/1.1.
PATH_INFO	Additional path information, as given by the client. This consists of the trailing part of the URL after the script name, but before the query string, if any.
PATH_TRANSLATED	A translated version of PATH_INFO that takes the path and performs any necessary virtual to physical mapping.
QUERY_STRING	Query information stored in the string following the question mark (?) in the HTTP request.
REMOTE_ADDR	The IP address of the client or agent of the client (for example, gateway or firewall) that sent the request.
REMOTE_HOST	The host name of the client or agent of the client (for example, gateway or firewall) that sent the request. If the Web server does not have this information, it will set REMOTE_ADDR and leave this empty.
REMOTE_USER	This contains the user name supplied by the client and authenticated by the server.
REQUEST_METHOD	The HTTP request method such as POST and GET.
SCRIPT_NAME	A virtual path to the script being executed. This is used for self-referencing URLs.
SERVER_NAME	The server's host name, or IP address, as it should appear in self-referencing URLs.
SERVER_PORT	The TCP/IP port on which the request was received.
SERVER_PROTOCOL	The name and version of the information retrieval protocol relating to this request. This is normally HTTP/1.0.
SERVER_SOFTWARE	The name and version of the Web server under which the application is running.

Data Out

Once the application has completed processing, it is time to return an HTML response to the client browser via the Web server. This is where the **HTTPCgiContext** class saves a lot of work over traditional CGI applications. In applications prior to CA-Visual Objects 2.5, a complete HTML document needed to be generated and sent to the Web server with the **FWrite**() function using the StdOut handle. With CA-Visual Objects 2.5 we simply use the **Write**() method of the **HTTPCgiContext** class as shown in the following function:

```
FUNCTION GreetEm

	LOCAL oCGI   AS HTTPCGIContext
	LOCAL cHtml  AS STRING
	LOCAL sFName AS STRING
	LOCAL sLName AS STRING

	oCGI := HTTPCGIContext{}

	sFName := oCGI:GetParamValue(“First”)
	sLName := oCGI:GetParamValue(“Last”)

	cHtml := oCGI:HTTPMsg("Welcome to our world, " + sFName + “ “ + sLName)

	oCGI:Write(cHtml)
	oCGI:Close()

RETURN
```

As you can see, CGI programming has gotten much simpler with the new functionality provided by CA-Visual Objects 2.5.

ISAPI

The Microsoft Internet Server Application Programming Interface (ISAPI) was introduced due to the inefficiencies of traditional Common Gateway Interface (CGI) applications. Under the CGI paradigm, the server created a separate process for each processing request received. The more concurrent requests that are received, the more processes created by the server. Creating a process for every request is not only time-consuming but requires large amounts of server RAM. The CGI technique is fine on a low traffic site, but the memory limitations start to slow down server operations. Therefore, scalability of CGI applications limits their usefulness.

One way to avoid this problem is to convert the CGI executable file into a DLL. The server loads the DLL the first time a request is received and the DLL then stays in memory, ready to service other requests until the server decides it is no longer needed. To facilitate the conversion of CGI executable files into DLLs, Microsoft released the ISAPI specification. This specification provides a framework (runtime dynamic linking) that is known to the server, which can then use the **LoadLibrary** and **GetProcAddress** functions to retrieve the starting address of the DLL.

DLLs which conform to this framework, also called ISAPI extensions, are loaded at runtime by the HTTP server and are called at the common entry points of **GetExtensionVersion** and **HttpExtensionProc**.

ISAPI extension DLLs are loaded into the same address space as the Internet server which means that all the resources made available by the Internet server process are also available to the ISAPI DLLs. There is minimal overhead associated with executing these applications because there is no additional overhead for each request. In addition, in-process applications scale much better under heavy server load conditions.

Besides the ISAPI extension, which emulates the traditional CGI process, the Microsoft specification provides for the creation of an ISAPI filter. An ISAPI filter is a replaceable, dynamic-link library (DLL) which the server calls whenever there is an Hypertext Transfer Protocol (HTTP) request. When the filter is first loaded, it registers with the server what sort of notifications will be accepted. After that, whenever a registered event occurs, the filter is called upon to process the event. When used with the Microsoft Internet Information Server (IIS), many custom schemes may be created including encryption, decryption, enterprise object instantiation, authentication, URL mapping, CGI-type servicing and logging. We will not be covering ISAPI filters in this session.

ISAPI Extensions

Creating ISAPI extensions prior to CA-Visual Objects 2.5 was a long and laborious undertaking. The addition of the **HTTPExtensionContext** class into this version has made the creation of these extensions truly possible. Before we launch into an example, let's examine the ISAPI extension specification.

As we mentioned earlier, ISAPI extensions are linked dynamically at runtime by the server through the Win32 API **LoadLibrary** call. Every ISAPI extension must define and export two entry points: **GetExtensionVersion** and **HTTPExtensionProc**. The function pointers for the entry points are retrieved by the server through the **GetProcAddress** API.

Unlike CGI application, ISAPI extensions are loaded in the same address space as the server by default. This is a configurable parameter in IIS 4.0 and 5.0. When loaded into the same address space, all of the resources that are made available by the server process are also available to the ISAPI extension. Minimal overhead is needed to execute the extensions because there is no additional overhead for each request.

Loading an ISAPI extension is typically faster than starting a separate executable. Context and task switching are reduced to a minimum since the extension is loaded into the memory space of the server. The same extension is also capable of servicing multiple requests which makes ISAPI extensions more scalable.

When the server initially loads the extension, it calls the **GetExtensionVersion** entry point to retrieve the version number of the specification on which the extension is based. **GetExtensionVersion** also provides for a short text description of the extension, which is useful for server administrators.

CA-Visual Objects 2.5 implements this entry point with the following defines and function:

```
DEFINE   HSE_VERSION_MAJOR          :=  2      // major version of this spec
DEFINE   HSE_VERSION_MINOR          :=  0      // minor version of this spec

FUNC GetExtensionVersion( pVer AS _WINHSE_VERSION_INFO)    AS LOGIC PASCAL

    EnterCriticalSection(@pCS)

    pVer.dwExtensionVersion := MAKELONG (HSE_VERSION_MINOR, HSE_VERSION_MAJOR)
    MemCopy(@pVer.lpszExtensionDesc[1], PTR(_CAST,DLL_DESC), SLen(DLL_DESC) )

    LeaveCriticalSection(@pCS)

    RETURN .T.
```

GetExtensionVersion accepts one input parameter, a pointer to the _WINHSE_VERSION_INFO structure:

```
STRUCT _WINHSE_VERSION_INFO
    MEMBER      dwExtensionVersion                          AS DWORD
    MEMBER DIM  lpszExtensionDesc[HSE_MAX_EXT_DLL_NAME_LEN] AS BYTE
```

This structure has two member variables: dwExtensionVersion, which holds the ISAPI specification version, and lpszExtensionDesc, which holds the description.

Interaction between the Web server and the ISAPI extension is conducted via a structure known as the Extension Control Block (ECB). All client requests are serviced through the **HttpExtensionProc** entry point. All information needed by the extension is passed to the extension via the ECB. The extension gets the most commonly needed information, such as the query string, path information, method name, and translated path, via the ECB.

You can think of the **HttpExtensionProc** entry point as the start function for your application. This is where your ISAPI extension will be called by the Web server. A sample **HttpExtensionProc** function is displayed below:

```
FUNC HttpExtensionProc( pECB AS _WINEXTENSION_CONTROL_BLOCK)    AS DWORD PASCAL
    //
    //  Entry point for All requests
    //
    LOCAL cHtml         AS STRING
    LOCAL nRet          AS DWORD
    LOCAL oISAPI        AS StdHTTPContext
    LOCAL oOldError
    LOCAL oError

    EnterCriticalSection(@pCS)

    oISAPI := StdHTTPContext{pECB}

    IF oISAPI:Error > 0
        nRet := oISAPI:Error
    ELSE
        oISAPI:StatusCode := 0

        oOldError := ErrorBlock( {|oErr| __MyError(oErr)} )
```

```
        BEGIN SEQUENCE
            cHtml := oISAPI:HTTPResponse()
        RECOVER USING oError
            cHtml := oISAPI:HTTPErrorMessage(oError)
            nRet := HSE_STATUS_ERROR
            oLogFile := LogFile{}
            oLogFile:DumpError(oError)
        END SEQUENCE

        ErrorBlock( oOldError )

        IF SLen(cHtml) > 0
            //
            //  Standard Behavior, return a HTML
            //
            IF oISAPI:Write(cHtml)
              nRet := HSE_STATUS_SUCCESS
          ELSE
                nRet := oISAPI:StatusCode
           ENDIF
        ELSE
            //
            //  User did his own processing
            //
            oISAPI:StatusCode := 200      // 200 OK
            nRet := HSE_STATUS_SUCCESS        //oISAPI:StatusCode
        ENDIF
    ENDIF

    oISAPI := NULL_OBJECT

    LeaveCriticalSection(@pCS)

    RETURN nRet
```

It is within this function that we instantiate the **StdHTTPContext** class, which inherits from the CA-Visual Objects 2.5 **HTTPExtensionContext** system class. This class is instantiated with the ECB as a single argument. The ECB is defined as a data structure displayed below:

```
STRUCT _WINEXTENSION_CONTROL_BLOCK
    MEMBER      cbSize                          AS DWORD    // size of this struct.
    MEMBER      dwVersion                       AS DWORD    // version info of this spec
    MEMBER      ConnID                          AS PTR      // Context number not to be
modified!
    MEMBER      dwHttpStatusCode                AS DWORD    // HTTP Status code
    MEMBER DIM  lpszLogData[HSE_LOG_BUFFER_LEN] AS BYTE     // null terminated log info
specific to this Extension DLL
    MEMBER      lpszMethod                      AS PSZ      // REQUEST_METHOD
    MEMBER      lpszQueryString                 AS PSZ      // QUERY_STRING
    MEMBER      lpszPathInfo                    AS PSZ      // PATH_INFO
    MEMBER      lpszPathTranslated              AS PSZ      // PATH_TRANSLATED
    MEMBER      cbTotalBytes                    AS DWORD    // Total bytes indicated from
client
    MEMBER      cbAvailable                     AS DWORD    // Available number of bytes
    MEMBER      lpbData                         AS BYTE PTR // pointer to cbAvailable bytes
    MEMBER      lpszContentType                 AS PSZ      // Content type of client data
    MEMBER      pGetServerVariable              AS GetServerVariable     PTR
    MEMBER      pWriteClient                    AS WriteClient           PTR
    MEMBER      pReadClient                     AS ReadClient            PTR
    MEMBER      pServerSupportFunction          AS ServerSupportFunction PTR
```

This control block contains the following fields:

CbSize(IN)	The size of this structure.
DwVersion(IN)	The version information of this document. The HIWORD has the major version number and the LOWORD has the minor version number.
ConnID(IN)	A unique number assigned by the HTTP server and which should not be modified.
DwHttpStatusCode (OUT)	The status of the current transaction when the request is completed.
LpszLogData (OUT)	Buffer of size HSE_LOG_BUFFER_LEN. Contains a null-terminated log information string, specific to the ISAPI applications, of the current transaction. This log information will be entered in the HTTP server log. Maintaining a single log file with both HTTP server and ISAPI application transactions is very useful for administration purposes.
LpszMethod (IN)	The method with which the request was made. This is equivalent to the CGI variable REQUEST_METHOD.
LpszQueryString (IN)	A null-terminated string containing the query information. This is equivalent to the CGI variable QUERY_STRING.
LpszPathInfo (IN)	A null-terminated string containing extra path information given by the client. This is equivalent to the CGI variable PATH_INFO.
LpszPathTranslated (IN)	A null-terminated string containing the translated path. This is equivalent to the CGI variable PATH_TRANSLATED.
CbTotalBytes (IN)	The total number of bytes to be received from the client. This is equivalent to the CGI variable CONTENT_LENGTH. If this value is 0xffffffff, then there are 4 gigabytes or more of available data. In this case, **ReadClient** should be called until no more data is returned.
CbAvailable (IN)	The available number of bytes (out of a total of cbTotalBytes) in the buffer pointed to by lpbData. If cbTotalBytes is the same as cbAvailable, the lpbData variable will point to a buffer that contains all the data as sent by the client. Otherwise, cbTotalBytes will contain the total number of bytes of data received. The ISAPI applications will then need to use the callback function **ReadClient** to read the rest of the data (beginning from an offset of cbAvailable).
LpbData (IN)	This points to a buffer of size cbAvailable that has the data sent by the client. You will get the first 48 KB of data.
LpszContentType (IN)	A null-terminated string containing the content type of the data sent by the client. This is equivalent to the CGI variable CONTENT_TYPE.

The majority of these members are exposed via access methods within the **HTTPExtensionContext** class.

After the **StdHTTPContext** class is instantiated, some error checking is performed and a call to the **HTTPResponse** method is performed. This is the method where we will actually perform our processing. Within this method we can either send responses directly to the client or accumulate them and return them to **HTTPExtensionProc**. If an accumulation is returned, **HTTPExtensionProc** takes care of sending it to the client. Once processing is complete, **HTTPExtensionProc** terminates with one of the following return values:

HSE_STATUS_SUCCESS	The execution has finished processing. The server can disconnect and free allocated resources.
HSE_STATUS_SUCCESS_AND_KEEP_CONN	The extension has finished processing and the server should wait for the next HTTP request if the client uses persistent connections. The extension should return this only if it was able to send the right content-length header to the client. The server does not have to keep the session open. The extension should return this value only if it has sent a connection: keep alive header to the client.
HSE_STATUS_PENDING	The extension has queued the request for processing and will notify the server when it has finished.
HSE_STATUS_ERROR	The extension has found an error while processing the request. The server can disconnect and free allocated resources.

Server Variables

While working in the **HTTPResponse** method, the application may have a need for various pieces of information about the Web server, client and method of calling. This information is available via the **HTTPExtensionContext**:**GetServerVariable** method. This information includes, but is not limited to, the following values:

AUTH_TYPE	The type of authentication used. For example: if basic authentication is used, the string is “basic”. For NT challenge-response, it is “NTLM”. Other authentication schemes have other strings. Since new authentication types can be added to the server, they can’t all be listed here. If the string is empty, no authentication is used.
CONTENT_LENGTH	The number of bytes that the script can expect to receive from the client.
CONTENT_TYPE	The content type of the information supplied in the body of a POST request.
PATH_INFO	Additional path information, as given by the client. This consists of the trailing part of the URL, after the script name but before the query string, if any.
PATH_TRANSLATED	The value of PATH_INFO, but with any virtual path name expanded into a directory specification.
QUERY_STRING	The information that follows the ? in the URL that referenced this script.
REMOTE_ADDR	The IP address of the client or agent of the client – for example, the gateway that sent the request.

REMOTE_USER	The user name supplied by the client and authenticated by the server. This comes back as an empty string when the user is anonymous (but authenticated).
UNMAPPED_REMOTE_USER	The user name before any ISAPI filter mapped the user, making the request to an NT user account. (The NT user account appears as REMOTE_USER).
REQUEST_METHOD	The HTTP request method.
SCRIPT_NAME	The name of the script program being executed.
SERVER_NAME	The server's host name or IP address, as it should appear in self-referencing URLs.
SERVER_PORT	The TCP/IP port on which the request was received.
SERVER_PORT_SECURE	A string of either 0 or 1. If the request is being handled on the secure port, this is 1. Otherwise, it is 0.
SERVER_PROTOCOL	The name and version of the information retrieval protocol relating to this request. This is normally HTTP/1.0.
SERVER_SOFTWARE	The name and version of the Web server under which the ISAPI extension DLL is executing.
ALL_HTTP	All HTTP headers that were not already parsed into one of the previous variables. These variables are of the form HTTP_<header field name>. The headers consist of null-terminated strings with the individual headers separated by line feeds.

HTTP_ACCEPT	A special case HTTP header. Values of the Accept: fields are concatenated and separated by a comma.
URL	Contains the base portion of the URL.

As you can see, these server variables are very similar to those available during CGI processing.

ISAPI Example

Now that we have looked at what goes into an ISAPI extension, let's examine a simple example. This example displays advertising banners similar to those seen at the top of Web pages.

The banner graphic files are stored in the images directory on our Web server. Each time the Web page is called, a different banner is displayed. The **HTTPResponse** method for this application is displayed below:

```
GLOBAL giCounter    AS INT

METHOD  HTTPResponse()              CLASS StdHTTPContext
    LOCAL cRet := "" AS STRING
    LOCAL aBanners   AS ARRAY

    // This Code could be moved into a database or external file for easy maintenance.
    aBanners := ArrayCreate(4)
    aBanners[1] := SELF:GetServerVariable("PATH_TRANSLATED")+"\images\ad_1.gif"
    aBanners[2] := SELF:GetServerVariable("PATH_TRANSLATED")+"\images\ad_2.gif"
    aBanners[3] := SELF:GetServerVariable("PATH_TRANSLATED")+"\images\ad_3.gif"
    aBanners[4] := SELF:GetServerVariable("PATH_TRANSLATED")+"\images\ad_4.gif"

    giCounter++
    IF giCounter > Len( aBanners )
        giCounter := 1
    ENDIF

    SELF:WriteImage( aBanners[giCounter] )

RETURN cRet
```

This is fairly simple and demonstrates the ease with which dynamic Web content can be created with CA-Visual Objects 2.5. We begin by defining a global variable to hold our position between invocations of the application. Within the **HTTPResponse** method we define an array of banners located in the images directory; and then we simply increment the global variable, check if we have incremented beyond the bounds of our array, and reset the counter if so. Once we know which banner to display, we call the **WriteImage** method with the local path to the banner file and then exit. To invoke banner.dll, we embed the following code in our HTML document:

```
<p align="center"><img src="/scripts/banner.dll”> </p>
```

This HTML code centers the banner and calls our banner.dll ISAPI application. Pressing the refresh button of the browser causes banner.dll to be re-invoked and the next banner in the array to be displayed.

This works great to simply display the banners, but most of the banners on the Internet display advertising for other Web sites and allow the user to click on the banner to be forwarded to the site. Since we are using an Image tag to display our banners, the only thing that can be displayed is an image. To remedy this situation and allow for the inclusion of an “href” tag, we need to turn to Server Side Includes. We will call our enhanced banner.dll with the following HTML code:

```
<p align="center"><!--#exec cgi="/scripts/banner1.dll?" --></p>
```

The code includes an #exec tag, which calls our CGI application called banner1.dll. The **HTTPResponse** method has been enhanced to include another array which contains the URLs that will be called when clicking on the banner image. We use the giCounter global to track our position in both of the arrays to synchronize them both. We use three **WriteClient** method calls to output the URL and the image tag which calls the image file. Also, notice that since we are actually outputting the contents of the image file, we need to modify the contents of the aBanners array to use relative paths:

```
GLOBAL giCounter    AS INT

METHOD  HTTPResponse()              CLASS StdHTTPContext
    LOCAL cRet := "" AS STRING
    LOCAL aBanners   AS ARRAY
    LOCAL aLinks     AS ARRAY

    // This Code should be moved into a database or external file for easy maintenance.
    aBanners := ArrayCreate(4)
    aBanners[1] := "/images/ad_1.gif"
    aBanners[2] := "/images/ad_2.gif"
    aBanners[3] := "/images/ad_3.gif"
    aBanners[4] := "/images/ad_4.gif"

    aLinks := ArrayCreate(4)
    aLinks[1] := "http://ad.bureau.com/jump.dll?ad_1.gif"
    aLinks[2] := "http://ad.bureau.com/jump.dll?ad_2.gif"
    aLinks[3] := "http://ad.bureau.com/jump.dll?ad_3.gif"
    aLinks[4] := "http://ad.bureau.com/jump.dll?ad_4.gif"

    giCounter++
    IF giCounter > Len( aBanners )
        giCounter := 1
    ENDIF

    SELF:WriteClient( '<A HREF=' + aLinks[giCounter] + '>' )
    SELF:WriteClient( '<img src="' + aBanners[giCounter] + '">' )
    SELF:WriteClient( '</A>' )

RETURN cRet
```

Additional improvements to this application include placing the image paths and URLs in an external file or database to provide for easier maintenance of the application, as well as tracking the number of times a banner is displayed for billing purposes.

Active Server Pages

Active Server Pages (ASP) are a server extension developed by Microsoft Corporation. ASP is a server-side scripting environment which provides an open, compile-free application environment in which HTML pages, scripts, and ActiveX server components may be combined to create powerful Web-based business solutions. When scripts are run on the server rather than on the client, the Web server performs all of the processing involved in generating the Hypertext Markup Language (HTML) pages that are returned to the client browser.

What Is an .Asp File?

Active Server Pages (ASP) are built around files with the file name extension .asp. An .asp file is a text file that can contain any combination of the following:

- Text
- HTML tags
- Script commands

A script command instructs the server to do something, such as assign a value to a variable.

It's easy to create an .asp file: Just rename any HTML file, replacing the existing .htm or .html file name extension with .asp. To make the .asp script file available to Web users, save the new file in a Web publishing directory. (Be sure that the associated virtual directory has Execute permissions enabled.) When the file is viewed in a browser, ASP processes the document and returns HTML, just as before.

CA-Visual Objects 2.5 and ASP

CA-Visual Objects 2.5 provides extensive support for ASP through the creation of ASP components. There are various classes provided that allow interaction with the client browser from within the components. These classes include:

IApplicationObject Class

The **IApplicationObject** class allows an ASP application to share information among all users of that application. In the world of ASP, an ASP application is comprised of all of the .ASP files in a virtual directory and its subdirectories. An example of sharing information might be where an ASP application contains a value for Invoice Number and must be shared by all of the ASP files within the application. When a new Invoice is generated, the ASP component written in CA-Visual Objects 2.5 could access that value, increment it and write it back out. From that point on, all other components could access the new Invoice number. Because the **IApplicationObject** object can be shared by more than one user, **_Lock**() and **_Unlock**() methods are provided.

IRequest Class

The **IRequest** Class allows retrieval of the values that the client browser passed to the Web server during an HTTP request. Once instantiated, this class stores the various values into an internal series of collections. These collections include:

- QueryString
- Form
- Cookies
- ClientCertificate
- ServerVariables

The **IRequest** class provides various methods for retrieving values from these collections. These methods can be broken into three basic categories:

- Methods used to retrieve the collections as arrays containing the name/value pairs {name, value} for each collection. An example of this category of method is **GetForms**(). This method is useful if you need all of the name/value pairs for a particular collection.
- Methods used to retrieve a certain value contained in one of the collections. The **GetForm**("fname") method is an example of this category.
- Methods which may be used to retrieve a value without knowledge of which collection it is contained within. The **GetItem**() method is an example of this category. It is important to note when invoking this category of methods that the collections are searched in the following order: QueryString, Form, Cookies, ClientCertificate, and ServerVariables. The methods return only the first value that they encounter, so care should be taken in the design of the application to prevent duplicate names.

It is important to note that an instance of the IRequest object is automatically created by the **IScriptingContext:OnStartPage**() method when the appropriate .ASP file is loaded by IIS. This relieves your program from having to manually create this object. The information stored in the collections may be accessed via the **IScriptingContext:Request** property.

IResponse Class

This class is used to return output to the client. This class provides a great deal of granularity for generating output to the client. The application can write strings to the HTTP output, add headers to an HTTP response, append messages to the Web server log file and even redirect the client browser to a different URL.

Like the **IRequest** object, an instance of the **IResponse** object is automatically created by the **IScriptingContext:OnStartPage()** method when the appropriate .ASP file is loaded by IIS. The application can gain access to this class via the **IScriptingContext:Response** property.

IScriptingContext Class

This class allows the retrieval of the built-in Active Server Pages (ASP) objects. The following accesses are available to the your application:

- Application Access – An **IApplicationObject** object instance.
- ASPApp Access – An **ASPAppAbstract** object instance.
- Request Access – An instance of the **IRequest** object.
- Response Access – An instance of the **IResponse** object.
- Server Access – An instance of the **IServer** object.
- Session Access – An instance of the **ISession** object.

There are also various methods within this class which generate HTML code automatically:

- **HTTPErrorMessage**()
    - Creates HTML code to display all values of a CA-Visual Objects 2.5 error object.
- **HTTPQueryString**()
    - Creates HTML code to display all query string items.
- **HTTPServerVariables**()
    - Creates HTML code to display all environment variables supplied by the Web server.
- **WriteImage**()
    - Creates HTML code to display images for ASP components.

IServer Class

The **IServer** class provides utility methods for use on the server. There are three methods within this class that are very useful when constructing ASP components:

- **_HTMLEncode**()
    - This method applies HTML encoding to a specific string. HTML encoding converts the string to a format which the client browser interprets. An example of HTML encoding would be **_HTMLEncode**("<b>"), which would return <b> to the client browser.
- **_MapPath**()
    - This method allows the component to map the specified relative or virtual path to the corresponding physical directory on the server. This is very useful for your component, which often needs a full path in order to access files such as a database. For example, the following line of code:

      ```
      cPath := SELF:Server:_MapPath( "/cgi-bin" )
      ```

      would store *C:/inetpub/scripts* into the variable *cPath*.

- **_URLEncode**()
    - The **_URLEncode**() method is used to apply URL encoding to a string. You may recall that when a CGI or ISAPI application receives the name/value pairs from the Web server, they are URL-encoded. This function allows you to apply the same encoding scheme on a string. For example, the following line of code:

      ```
      cEncode := SELF:Server:_URLEncode( "CA-Visual Objects 2.5")
      ```

      would store *Visual+Objects+2%2E5* into the variable *cEncode*.

ISession Class

This class is used to store information needed for a particular user session. Variables stored in the **ISessionObject** instance persist for the entire user session. The variables will not be discarded when the user switches between pages in the application. Methods of interest in this class include:

- **GetSessionID**()
    - This method is used to retrieve the value of a session's ID. This unique value allows an application to keep track of the user.
- **GetValue**()
    - This method allows the retrieval of a value stored in an instance of the ISessionObject object.
- **PutValue**()
    - This method allows the storage of a variable into an instance of the ISessionObject object.
- **Abandon**()
    - The Abandon() method destroys all of the objects stored in the current session and releases their resources.

IWriteCookie Class

The **IWriteCookie** class provides for altering the values and attributes of the cookies stored in the write-only cookie collection of the HTTP response. The class contains methods for setting all of the attributes of a cookie. Once written, cookies are returned to the Web server and, subsequently, the server application with all succeeding requests.

IReadCookie Class

The **IReadCookie** class provides for the retrieval of the values stored in cookies sent in an HTTP request.

ASP Example

As an example of an ASP component written in CA-Visual Objects 2.5, we have included a simple code fragment that returns a table containing all of the HTTP server variables:

```
GLOBAL goASPComponent      AS ASPComponent

PROCEDURE InitProc() _INIT 3
       goASPComponent := ASPComponent{}

CLASS ASPComponent INHERIT IScriptingContext
       PROTECT      cAction             AS STRING

METHOD HTTPResponse        CLASS ASPComponent
       LOCAL cRet          AS STRING
       DO CASE
       CASE SELF:cAction = "SERVERVARS"
              cRet := SELF:HTTPServerVariables()
       OTHERWISE
              cRet := "Unknown Action"
       ENDCASE
       RETURN SELF:Response:_Write(cRet)

ASSIGN Action(cNew)        CLASS ASPComponent
       SELF:cAction := SELF:GetVariantString(cNew)
       RETURN SELF:cAction

ACCESS Action              CLASS ASPComponent
RETURN SELF:cAction
```

After the component has been compiled, it can be called from an HTML page with the following code:

```
<a href="../JP115SN/response1.asp?Action=SERVERVARS">Display Server Environment
(via VO ASP Component)</a>
```

This line of HTML code simply calls the file response1.asp with an action of *SERVERVARS*. The contents of response1.asp are:

```
<html>

<p>
<h1>Response Page from my VO ASP Component</h1>

<%@ LANGUAGE = JScript%>

<%
oComp = Server.CreateObject("VoAspTest.Component")
oComp.Action = Request.QueryString("Action")
oComp.HTTPResponse()
%>

</html>
```

This instantiates a copy of our CA-Visual Objects 2.5 ASP component and sends the request to it. The output from this request is displayed in Figure 3:

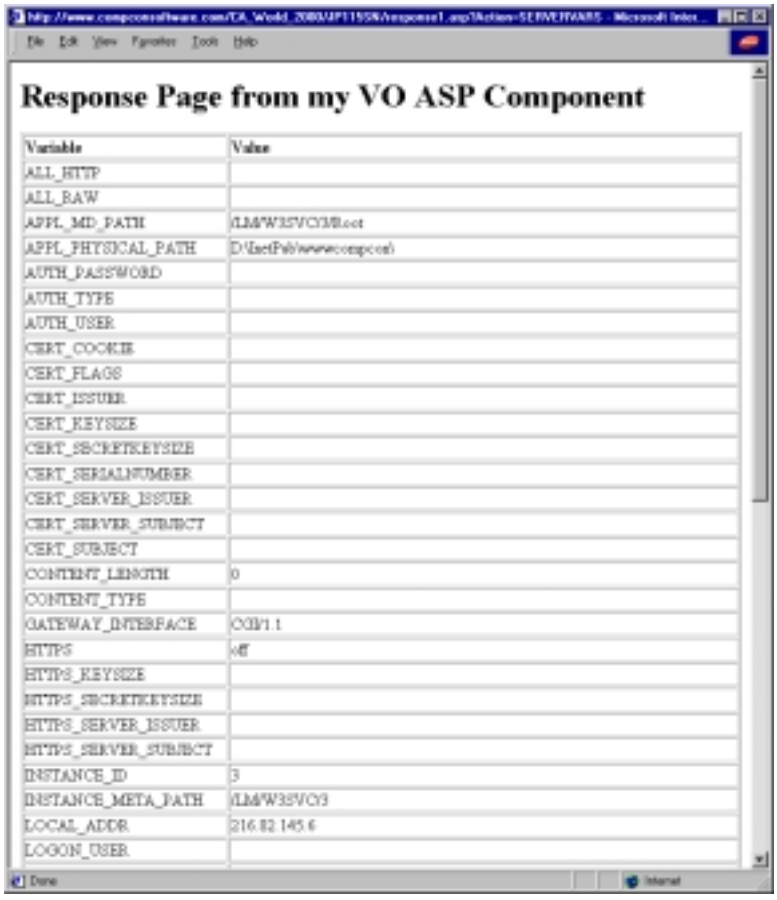

Figure 3: Server Variables Returned from an ASP Component

Conclusion

Creating dynamic Web content in CA-Visual Objects 2.5 is a very simple process as we have seen. The excellent collection of classes provided by CA-Visual Objects 2.5 easily handles the major hurdles. What you do with HTML Forms and the data that your users enter into them is limited only by your imagination. Use your imagination and have fun creating applications for the World Wide Web.

Bibliography

Eugene Eric Kim. *CGI Developer's Guide*. Sams.net Publishing, 1996.

Stephen Genusa. *Special Edition Using ISAPI*. Que Corporation, 1997.

David Fox. *HTML Web Publisher's Construction Kit.* Waite Group Press, 1995.

Brent Heslop and Larry Budnick. *HTML Publishing on the Internet for Windows*. Ventana Press, 1995.

Susan B. Peck and Stephen Arrants. *Building Your Own WebSite for Version 1.1*. O'Reilly & Associates, 1996.

Ian S. Graham. *HTML Sourcebook*. John Wiley & Sons, 1995.

David Taylor. *Creating Cool Web Pages with HTML*. IDG Books Worldwide, 1995.

David Fox. *HTML Web Publisher's Construction Kit.* Waite Group Press, 1995.

Tom Elledge is the lead Windows NT server administrator and a programmer/analyst with the US Department of the Interior - Bureau of Land Management in Reno, Nevada. He also serves as the technical Webmaster for the Nevada State Office of the BLM. He has been developing applications for Resource Management in CA-Clipper since the "Winter '85" release. In addition, he has been developing Internet applications in CA-Visual Objects since the pre-beta of this product. He has developed and delivered training to agency personnel across the country in the areas of UNIX, Internet and Informix SQL on behalf of the BLM National Training Center. Tom also owns and operates COMPCON Software, which develops and markets software for federal employees, and performs consulting services for the Internet and gaming industries. He can be reached via the Internet at telledge@compconsoftware.com.