# Get On The Web With Delphi 3

## *Everything you need to know about writing web applications using Delphi 3 (but didn't know you needed to ask): Part 1*

*by John O'Connell*

A rather ambitious subtitle, I know, but there's a lot to writing web server applications with the new Delphi 3.0 classes and components which isn't covered in the manuals. Perhaps you've thought about trying your hand at writing web server apps? In Issues 24 and 25 Bob Swart covered the basics of writing web applications using Delphi 3.0's new components, here I'll endeavour to describe what else you'll need to know but obviously I can't promise that I'll have covered *absolutely* everything else you'll need to know in this article and the follow-up which will come next month! All of the demo applications and web pages mentioned here are on this month's disk.

By way of getting started let's look at what a web server application actually is and how one generally works: I'm assuming you know a little about how web server applications work and that you're familiar with terms such as HTTP, MIME types and URL as used in relation to the internet/intranet, if not then take a look at previous articles by Bob Swart (Issue 15) and Steve Troxell (Issue 16) which discuss writing CGI and WinCGI web applications and some of the associated terminology.

The next four sections are essentially a web server applications primer which distills what has previously been discussed about CGI, WinCGI and ISAPI in previous articles.

A web server application (or just web application) executes on a web server in response to a request from a web browser which specifies the URL of the application it wants to execute. The application's output (which must be of some MIME type, such as `text/html`) is captured by the web server and delivered to the web browser via HTTP. In fact all communication between the web browser and web server is via HTTP, the protocol of the web. In HTTP terms a browser sends an *HTTP client request* to the server which sends back an *HTTP server response* the content of which can be a web page or the output of a web application specified in the request's URL. The output of a web application is sometimes referred to as a dynamic response, or a virtual or dynamic web document. Figure 1 shows a typical request header as sent to a server and the response header sent back from the server.

### Gateway Interfaces

All communication between the web server and web application takes place via a gateway interface, the most common of which is the Common Gateway Interface or CGI.

The first task of a web server gateway interface is to pass details of the request (such as the browser name, the list of MIME types acceptable to the browser, the IP address of the request sender, any request data and perhaps its length, the request method, the version of the HTTP protocol used and so on) to the web application. CGI parses or translates the received request header (Figure 1) into the individual request details which are accessed by the web application through the use of environment variables such as `REQUEST_METHOD`, `QUERY_STRING` and `HTTP_USER_AGENT` to name but a few. Delphi 3.0's `TWebRequest` class allows you to access request details as object properties.

The other task of the gateway interface is to capture the output of the web application and package it as a response to be sent to the browser which sent the request. The content of the response must be of some MIME type recognised by the browser.

CGI web applications must be console applications, but not all Windows RAD tools are capable of creating console applications. One very well known visual RAD tool springs to mind, but I'll leave you to guess its name! WinCGI, developed by Robert Denny, the author of O'Reilly's WebSite web server, is a variation of CGI designed to allow non-console web applications to participate in the web server applications game. With WinCGI the request is packaged as an INI file containing the request details as section/key values which can be read by the web application which then writes any response content

➤ *Figure 1:*
*Left column: A typical HTTP request header*
*Right column: a typical HTTP response header*

```
GET /cgi-bin/webapp.exe?custname=Jaimie+Sach HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0 (Win95; I)
Accept: image/gif, image/x-xbitmap, image/jpeg,
image/pjpeg, */*
From: user@acompany.com
```

```
HTTP/1.0 200 OK
Date: Monday, 21st July-96 15:10:00 GMT
Server: WebSite/1.1e
Content-type: text/html
Content-length: 51
<HTML><BODY><P>Hello
Jaimie+Sach</P></BODY></HTML>
```

to an output file named in the request details. Because of the additional overhead of creating the request INI file and response output file, WinCGI is a bit slower than CGI but then, borrowing from an old adage, a slow interface is better than no interface at all. Despite having talked about WinCGI, there's no good reason to write WinCGI applications with Delphi, which can produce standard CGI applications which are faster and more efficient than their WinCGI counterpart. However WinCGI applications are much easier to debug.

More recent gateway interfaces are based on web server APIs: Netscape's NSAPI and Microsoft's ISAPI, the latter becoming more and more common. A web server which supports ISAPI calls an ISAPI web application DLL via a predefined entry point function in which the request is processed and the response generated. An ISAPI web application is a DLL in which request details or request data are accessed via a parameter (called an Extension Control Block) which is passed to the entry point function (named `HTTPExtensionProc`); response content is simply passed as a parameter to a callback function (`ServerSupportFunction` or `WriteClient`) provided by the ECB, all of which means that data transfer is very fast compared with CGI/WinCGI. The ISAPI2 unit included with Delphi defines the data-types and functions used to write ISAPI DLLs. A more detailed discussion of ISAPI can be found in Steve Troxell's article in Issue 19.

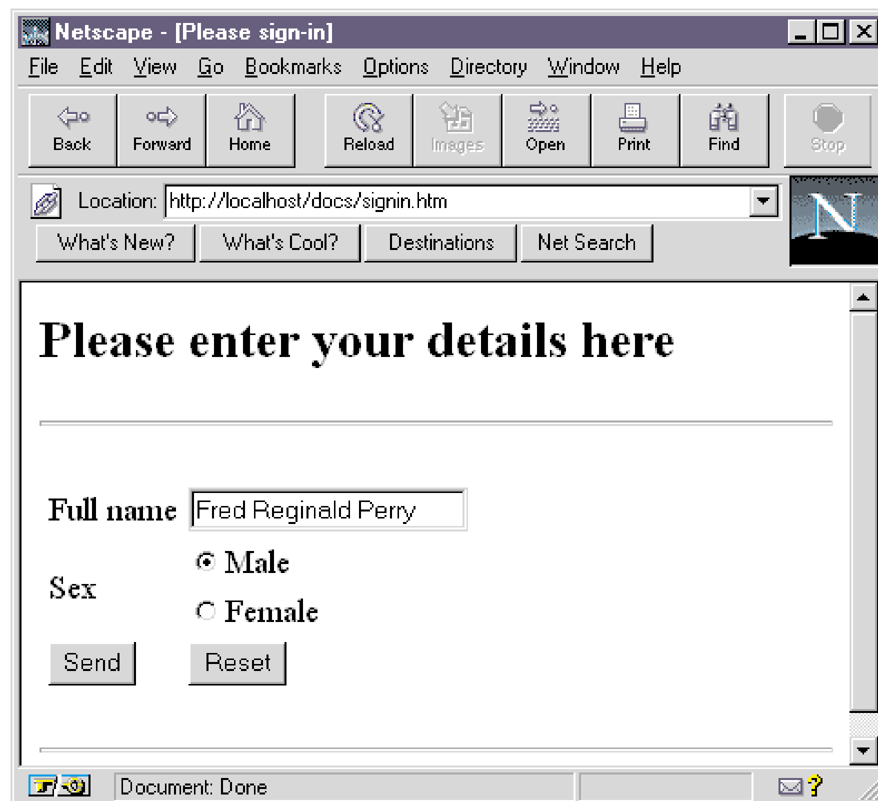
## Server API Or CGI?

Besides speed, the major advantage of server APIs over CGI or WinCGI is one of efficiency: server API web applications are actually DLLs which are loaded once and whose entry point is executed once for each client request. Contrast this with CGI and WinCGI web applications which are separate executables which must be loaded and unloaded for each client request thus generating quite a processing overhead. Not surprisingly ISAPI and NSAPI web applications are becoming increasingly popular in the world of 32-bit Windows web servers.

However, there is a downside to server APIs because safe, reliable server API web applications are more difficult to write due to thorny issues such as thread handling which must be dealt with. Because only one instance of the DLL can be loaded into the web server's process space and the DLL entry point function must return as soon as possible in order that other queued requests can be serviced, individual threads are spawned by the web server to service each client request: in other words any code in the entry point function will most certainly be called by different threads which necessitates writing thread-safe code, thus adding an extra level of complexity for the server API application programmer.

More importantly, because DLLs share the host application's process (and address) space, so an errant server API DLL can do more harm to the web server than a CGI or WinCGI application (which runs as a separate protected process) could ever inflict. It seems there's a price to pay for everything these days.

In short, much more care is needed writing server API DLLs. On the other hand, the operating system (assuming it's Unix or Win32) generally prevents CGI or WinCGI applications from inadvertently crashing the web server.




➤ *Figure 2: A typical HTML form*

## Request Methods...

...or, how request data is passed to the web application. The purpose of most web applications is to process data entered into an HTML form (presented by the browser) which contains basic common GUI data entry controls such as edit boxes, drop-down lists, radio buttons and checkboxes. At the press of a pushbutton the form's data is submitted as part of a client request to a specified web application which must generate and send back a response. Form data is passed to a web application by one of two ***request methods*** each of which have their own advantages and disadvantages as we'll see.

The `GET` request method appends form data to the web application's URL. Using the form

in Figure 2 as an example, after pressing the Submit button to send the form to the server the URL becomes:

```
http://localhost/cgi-bin32/
  showrequest.exe?fullname=
  Fred+Reginald+Perry
  &sex=M&sendfields=Send
```

We can see that the field data is separated from the URL of the web application by the question mark character and each form field's name=value pair is separated by the ampersand character (&). Notice that space characters within the form data are replaced with a plus character (+). This is URL encoding, which enables reserved characters to be included as part of a URL. Other reserved characters are encoded as their two digit ASCII hex code preceded by a percent character (%), so that my name would be John+O%27Connell when URL encoded.

Everything after the question mark in the above URL is known as a query string, which must be parsed by the web application into individual name=value field pairs. Figure 1 shows a GET request header.

A CGI web application can access the query string by reading the QUERY_STRING environment variable; a WinCGI application can access the form data by reading the QUERY STRING key in the [CGI] section of the INI file containing the request details; an ISAPI application (or Internet Server Application: ISAs as Microsoft like to call them) simply reads the lpszQueryString field of the ECB which is passed to the web application's exported HTTPExtensionProc function.

The POST request method differs from GET in that the form data is not appended to the URL but must be read separately by the web application. For a CGI application this means reading the data from the standard input stream. WinCGI conveniently parses the POSTed data into URL decoded name=value pairs placed in the [Form Literal] section of the request INI file. An ISAPI DLL reads POSTed data from ECB.lpbData and by calling the ECB.ReadClient function if more data needs to be read in addition to that in lpbData. The length of the POSTed data is contained in the CONTENT_LENGTH CGI variable or in ECB.cbTotalBytes for an ISA. Note that POSTed request data need not be URL encoded (after all it's not part of the URL) though some servers do so anyway.

A POST request header is similar to a GET request header:

```
POST /cgi-bin/webapp.exe HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0 (Win95; I)
Accept: image/gif, image/x-xbitmap,
  image/jpeg, image/pjpeg, */*
From: user@acompany.com
Content-Length: 128
```

except that the request data is not part of the header but is read by the web application which reads the request data whose length is indicated by the content length header item.

## Which Request Method?

The main advantage of using GET is its faster request data transfer. In addition, if, after executing the web application, the URL is bookmarked then because the query string is embedded within the bookmark URL, the web application will be executed with the saved query string when the bookmarked URL is revisited. And rather usefully the request query string can be entered directly as part of the URL which does away with the need for an HTML form and is useful for testing the web application.

The disadvantages of GET are its relative lack of security: anyone can peer over your shoulder and see sensitive data embedded in the URL of a GET web application, and a limited maximum query string length of around 1Kb, depending on the browser. Also the web application's URL can become quite cluttered and unsightly particularly with long query fields.

The advantages of using POST are the relative security and the fact that much more request data can be passed to the web application. Bear in mind the necessity for an HTML form though.

You'll find that most web search sites (such as Lycos and Yahoo) use GET.

## WebModules, WebRequests And WebReponses

The core logic of a Delphi 3.0 web application resides in the WebModule onto which you can drop the various non-visual components used to build a web application.

A WebModule is actually a DataModule which contains an additional built-in component, a WebDispatcher. The WebDispatcher is responsible for the flow of control in the web application as we'll see later. Although the WebModule has its own built-in WebDispatcher, a separate TWebDispatcher component is provided in the Internet tab of the component palette. This allows you to convert any DataModule to a WebModule, in fact you could add a TWebDispatcher to all your DataModules which could then be used as the main form of a web application project. Having a TWebDispatcher (or any other non-visual component from the Internet component page) in a DataModule used by other non-web applications won't have any side effects on those other applications, other than some increase in code size. Conversely you can use a WebModule as a DataModule in a GUI database application.

Note that only one WebDispatcher can exist in a web application: adding a TWebDispatcher to a WebModule is not allowed by Delphi and raises an exception.

If you examine the web application's project source you'll notice that it's a little different from that of your usual application: there's no sign of the Forms unit usually seen in the DPR file but instead the HTTPApp unit is included along with the WebModule's unit and another unit which defines the project's Application variable type. In the case of a CGI or WinCGI application, the Application variable's type will be TCGIApplication as defined in the CGIApp unit used by the project source; an ISAPI or NSAPI project's Application will be

of type TISAPIApplication as defined in the ISAPIApp unit. You may be wondering why there's no separate TWinCGI Application object type, well it's not needed because a console ({$APPTYPE CONSOLE}) TCGIApplication application is a CGI web application, whereas a GUI ($APPTYPE GUI}) TCGIApplication is a WinCGI application, obvious really.

## The WebDispatcher

As its name implies, TWebDispatcher dispatches things: to be precise it passes a TWebRequest object that represents the HTTP request and a TWebResponse object, the HTTP response, to one or more TWebActionItem(s), each of which has an OnAction event property whose handler will usually contain code to build the content for the response object. Essentially the TWebDispatcher controls the flow of execution of a Delphi web server application.

The TWebDispatcher's Actions property, of type TWebActionItems, contains the list of TWebActionItem objects which can each be set to handle a specific request type. Action items can be added or edited via the Actions property editor which behaves much like the Fields property editor found in TQuery/TTable and TStoredProc.

Each action item has properties (which are listed and described in Table 1) which define the type of request it will get to handle: the action items which get to handle the request are those whose MethodType and PathInfo properties match the TWebRequest's MethodType and PathInfo properties, or whose Default property is True, regardless of its PathInfo. An action item's MethodType of mtAny will match any TWebRequest.MethodType which means that only the action item's PathInfo property need match the TWebRequest's PathInfo in order for it to be selected to handle the request. Set an action item's MethodType to mtAny if you don't care which request type the action item can handle.

As it is possible for more than one action item to share the same PathInfo and matched MethodType property values, the work of building the response content may be shared among multiple OnAction event handlers, more on which later. The TWebActionItem's OnAction event is where the work of building the response for a particular request is performed. The OnAction event which is of type THTTPMethodEvent is defined as:

```
procedure (Sender: TObject;
  Request: TWebRequest;
  Response: TWebResponse;
  Handled: boolean);
```

Sender is the TWebActionItem for which the event is triggered: the Request and Response parameters are references to the TWebRequest and TWebResponse instances which are passed by the WebDispatcher to the OnAction event handler. To generate response content simply assign to Response.Content. As a simple example, the following OnAction handler causes "Hello world" to be displayed by the browser which sent the request:

```
procedure TWebModule1.WebModule1
  WebActionItem1Action(
  Sender: TObject;
  Request: TWebRequest;
  Response: TWebResponse;
  var Handled: Boolean);
begin
  Response.Content :=
    'Hello world';
end;
```

The final event parameter, Handled, determines whether the response has been handled and is ready to be sent to the server: this is very important where multiple action items are to be used to build the response content to a request. Setting Handled to True signals to the web dispatcher that the request has been handled, otherwise another OnAction event handler will be called to handle the request until Handled is set to True.

In an OnAction event the default value of Handled is True so it can usually be left alone, but setting it to False will trigger the OnAction event of the next chained action (if any) item and so on. Setting Handled to False in the default action item's OnAction handler causes no response content to be sent back to the browser, which will then display a "no response from server" error message.

The web dispatcher sandwiches calls to OnAction event handlers between it's own two event handlers, BeforeDispatch and AfterDispatch both of type THTTPMethodEvent, the OnAction event type.

➤ *Table 1: TWebActionItem properties*

| Property | Purpose |
|---|---|
| Default | If True the action item will be triggered if no other action items handle the request |
| Enabled | Enables or disables the action item |
| MethodType | Specifies the request method of TMethodType = (mtGet, mtPost, mtHead, mtPut, mtAny) which the Action item can handle |
| PathInfo | If the request URL contains the specified PATHINFO then this action item will be triggered to handle the request. The PATHINFO of a URL is the part of it preceded by a '/' and following the web application name. For example, the PathInfo of http://www.webserver/cgi-bin/mywebapp.exe/greeting is "/greeting" |
| OnAction | The event handler in which content can be assigned to the response |

| Web Application Type | Application Object Type | Request Object Type | Response Object Type |
| --- | --- | --- | --- |
| CGI application ($APPTYPE CONSOLE) | TCGIApplication | TCGIRequest | TCGIResponse |
| Windows CGI application ($APPTYPE GUI) | TCGIApplication | TWinCGIRequest | TWinCGIResponse |
| ISAPI or NSAPI Server API DLL | TISAPIApplication | TISAPIRequest | ISAPIResponse |

➤ *Table 2: Web application, request and response object types*

The application CHAINED.DPR demonstrates web action item chaining and the effects of various settings of the `Handled` parameter of the chained `THHTPMethodEvent` handlers. In this application each `OnAction` event handler appends the sender's name to the response content which ultimately is sent to the browser as soon as the last matching `OnAction` handler has handled the request, as passed to each action item chosen by the web dispatcher to handle the request type, and set the `Handled` parameter to `True`.

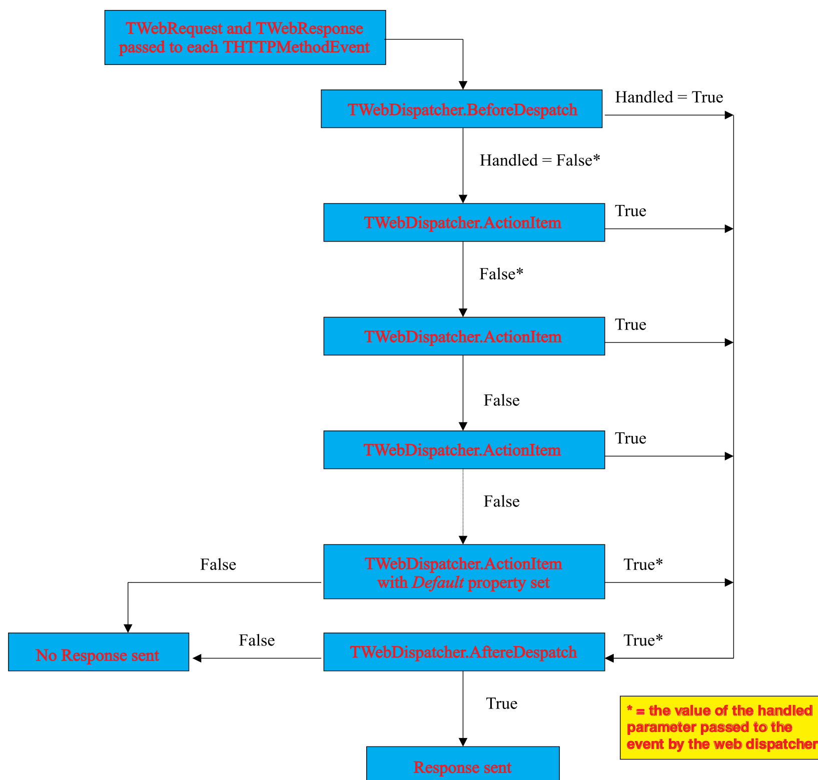
In order to try making things a bit clearer, let's look at the flow of a web application through the web dispatcher's action items.

Firstly, the `Application` object creates the `Request` and `Response` instances, each of the appropriate type such as `TCGIResponse` and `TCGIRequest` (see Table 2), which are then passed to the web dispatcher. Secondly, the web dispatcher's `BeforeDispatch` handler is called with the `Request` and `Response` passed as parameters and `Handled` set to `False`, this ensures that an `OnAction` event will be triggered to handle the request. Setting `Handled` to `True` will signal the request as handled and no `OnAction` events will be triggered in which case the `AfterDispatch` event handler will be called, provided no response content has been sent to the server (with a call to the `Response` object's `SendResponse` method). Then, assuming the request hasn't yet been handled, the `Request` and `Response` objects are passed to the `OnAction` handler of the first action item matched by its properties to handle the request and build the `Response` content.

If the `OnAction` handler leaves `Handled` set to `True` (the default) then provided no response has been sent, the web dispatcher calls its `AfterDispatch` event handler and no further action items are called, but if the `OnAction` handler sets `Handled` to `False` then the `Request` and `Response` are passed to the next matching action item's `OnAction` event until all matching action items are exhausted or until `Handled` is set to `True` by one of them. If at this stage the request hasn't been signalled as handled then the default action item's `OnAction` handler is called: if this doesn't set `Handled` to `True` then no `Response` content will be sent and the web application will end here, otherwise if `Handled` is set the `AfterDispatch` event is triggered, provided the response hasn't yet been sent. Next, the `AfterDispatch` event handler has it's `Handled` parameter set to `True`, setting it to `False` will signal that the request hasn't been handled. After all that, the final stage is performed by the `Application` object: if the request has been handled and the response hasn't yet been sent, then `Application` will call the response object's `SendResponse` method. If the request hasn't been handled then no response is sent. Figure 3 shows the flow of execution within the `WebModule` or the `WebDespatcher`.

Try experimenting with the CHAINED application (which is on the disk) yourself by changing the last line in any of the `OnAction` event handlers so that only one or two of the three chained matching action items are called.

➤ *Figure 3*



A few points of interest. The OnAction event of the default web action item may be called twice: once as part of the chain of multiple action item events and again if the request is still not handled after all chained action items have been called. Also, a web application can function without any action items because the request can be handled within the BeforeDispatch event handler.

The BeforeDispatch and AfterDispatch handlers can be useful for building the opening and closing parts of the Reponse such as the title and header tags of the content. And finally, the class type of the actual TWebRequest and TWebResponse instances depends on the type of application as summarised in Table 2.

## TWebRequest And TWebResponse

The TWebRequest and TWebResponse base classes encapsulate the HTTP request sent to the application and the HTTP response returned by the application, the application type-specific request and response classes (TCGIRequest/TCGIResponse etc) are all derived from these two base classes. The SHOWREQUEST demo application displays the properties of the WebDispatcher's Request instance: use the form in GUESTBK.HTM to see SHOWREQUEST working, or just append some URL encoded quer yfields to the web application's URL.

The values of a TCGIRequest's properties are obtained from the appropriate CGI environment variable, for a TWinCGIRequest the property values come from the request INI file and for a TISAPIRequest they're obtained from a combination of calls to the Extension Control Block's GetServerVariable function and the values of some of the ECB fields.

Table 3 lists some of the more useful TWebRequest properties and their purpose in identifying the request.

For a GET request the form fields are contained in the QueryFields string list property; for a POST request the form fields are contained in the ContentFields string list property. This makes accessing request data as easy as using the TStringList's Values property to retrieve field values by name. As we've seen, the PathInfo property is that part of the URL used to select which web dispatcher action item(s) is used to handle the request. UserAgent is very useful for customising response content to the browser's capability: the response content for a Netscape 2.0 (or later) client could contain HTML frame tags or even HTML scripting code which the browser can understand whereas the response content for other less-capable browsers could be just

➤ *Table 3: Useful TWebRequest properties*

| Property | Meaning |
| --- | --- |
| MethodType | The request method of TMethodType |
| ContentFields | TStrings list of parsed POSTed content fields |
| CookieFields | TStrings list of cookie fields |
| QueryFields | TStrings list of parsed GET query fields |
| Method | 'GET', 'PUT', 'POST' or 'HEAD' |
| ProtocolVersion | The HTTP protocol version (HTTP/1.0) |
| URL | The full URL of the request's target - the web application |
| Query | The URL-encoded GET query string |
| PathInfo | That part of the path after the application name eg PathInfo = 'switch' in the URL http://www.acme.com/script.exe/switch?field=value&field2=value |
| PathTranslated | The URL translated to a server relative path |
| Authorization | The HTTP authentication used to identify a user |
| Cookie | List of semi-colon delimited URL-encoded cookies |
| Accept | The list of MIME types that the client/browser can handle (derived from the Accept request header item) |
| From | The email address of the user making the request: not always supported by some browsers for security reasons |
| Host | The server host name |
| IfModifiedSince | User only wants information changed since this date |
| Referer | The URL of the web document which generated the request |
| UserAgent | The user's browser, such as Mozilla/3.0 for Netscape 3.0 |
| ContentEncoding | The encoding scheme for the content |
| ContentType | The MIME type of the POSTed content |
| ContentLength | The length in bytes of any POSTed content |
| RemoteAddr | The remote IP address of the user |
| RemoteHost | The remote host name of the user making the request |
| ScriptName | The path of the web application |
| ServerPort | The port number on which the server is running |

standard HTML, as long as it didn't use frames.

If your wondering about the purpose of the Cookie property, check out Bob Swart's article in October's issue where he uses cookies to maintain state within a web application. I'll talk more about cookies next month and discuss the various approaches to maintaining state between web pages and we sessions.

TWebRequest provides a number of useful methods: WriteClient and WriteString are used to send data back to the client, the method TWebResponse.SendResponse actually uses WriteString to achieve it's purpose. Conversely, ReadClient and ReadString read data from the client as is necessary for a POST request though you'll rarely need to do so. The ExtractFields method breaks down a delimited string into a string list, and is used by the methods ExtractContentFields, ExtractCookieFields and ExtractQueryFields which break down response content, cookie strings and query field strings into string lists. GetFieldByName retrieves a specified CGI request variable as a string.

As I've said, TWebResponse is the abstract base class for TCGIResponse, TWinCGIResponse and TISAPIResponse, in fact the first two classes are the very same: TWinCGIResponse is simply a class reference to TCGIResponse.

The most important and most used property of TWebResponse is Content which represents the data sent back to the client as we've already seen in previous examples. Similar to Content is the ContentStream property which specifies a stream from which the response content is derived. HTTPRequest identifies the web request for which the response must be provided.

Other properties are important for further defining the response. The numeric StatusCode property specifies the HTTP status code for a particular request. Status codes specify how the server (and browser) should treat the response, for example a statuscode of 500 will cause the browser

| Status code | ReasonString |
|---|---|
| 200 | OK |
| 204 | No content |
| 301 | Document permanently moved |
| 302 | Document temporarily moved |
| 401 | Unauthorised |
| 404 | Not found |
| 500 | Internal server error |

➤ *Table 4: Useful HTTP status codes and reason strings*

to display a "Server Error" message. A full list of HTTP status codes and their purpose or meaning are listed at www.w3c.com but I've listed a few of the more useful ones in Table 4. Associated with StatusCode is the ReasonString property which is a string description of the status code's meaning, but you don't have to set this every time you set StatusCode because it's done for you automatically: obviously this means that if you want to set your own ReasonString then do so after setting StatusCode! The default response status code of 200 indicates a normal response containing some content.

The ContentType string property allows the MIME type (default text/html) to be specified, the list of MIME types the client can handle are found in the Accept property of the Request object. The Expires TDateTime property specifies when the response will become out of date, this has an impact on the web page caching performed by some browsers; LastModified indicates the date and time the content was last changed at the server which is useful because a client can send a HEAD request (that is, TWebRequest.MethodType = mtHead) which just requests a response header *without* the content (thus avoiding a possibly wasteful transfer of a large amount of data), therefore by examining LastModified the client can decide to retrieve the response content using a subsequent GET request if a locally stored copy of the response content is older than the last modified date.

You don't have to specify values for each and every response property, the relevant defaults are set by the Request object's SendResponse method which builds the response header.

I mentioned TWebResponse.SendResponse whose purpose is to output a response header followed by whatever is assigned to the Content or ContentStream properties. other methods for sending response content include SendRedirect sends a redirection header as part of the response header: redirection headers simply point or redirect the client to retrieve the URL specified in the argument to this method; SendStream sends its TStream argument to the client but to work correctly SendResponse must be called beforehand to set up the response header; SetCookieField sends at least one cookie to the client. As promised we'll look at this method and at cookies in more detail a bit later.

One point worth noting about the ContentStream property is that you don't have to worry about freeing the assigned stream as the SendResponse method frees it for you.

## Until next month

So far we've looked in detail at how the WebModule and WebDispatcher work, how the request and response objects are passed to the dispatcher's event handlers, and how the path of these objects can be controlled from within the event handlers. Which concludes this month's discussion of the basic structure of a Delphi 3.0 web application.

Next month we'll take an in-depth look at ways of producing response content using the page producer classes and ways of saving state in web applications (Bob Swart covered these topics to some degree in October's article), as well as a detailed look at writing ISAPI and NSAPI web applications which use the BDE. I'll also show how to set up a local web-server (for free) for developing and testing your web applications.

Until then...

John O'Connell is a freelance software consultant/developer specialising in Delphi and database application development. Email him on 73064.74@compuserve.com.