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 2

by John O'Connell

Last month we looked at how a WebModule and WebDispatcher-based web application worked with the TWebResponse and TWebRequest objects which encapsulate an HTTP request and response. This month we'll take an in-depth look at what you need to know about using page producer components and creating your own custom page producers for use building the web application's response content. We'll also discuss how to configure Netscape servers and NSAPI for use with server API DLLs and what to watch out for when using the BDE in ISAPI and NSAPI applications. Finally, I'll include some other useful development tips, all aimed at maximising the capabilities of your web applications.

Page Producer Productivity

Whereas the WebDispatcher takes care of routing the Request and Response objects to the relevant TWebActionItem's OnAction event handler, page producers take care of generating the Response object's content from HTML templates or from datasets in HTML table form.

Bob Swart has already discussed page producers in previous articles, here I'll go into some more detail about how to make the most of the page producer components.

It is possible to chain together page producers such that the Content property of one can be assigned to the content property of another. This can be useful for incrementally building the response content, particularly where the content must be customised for the capabilities of different browsers. For example, we can take three TPageProducers, the first having it's HTMLDoc property contain various HTML-transparent (or custom) tags which can be expanded into yet more custom tags by the page producer's OnHTMLTag event. The first page producer's Content property can then be assigned to the second page producer's HTMLDoc property and the custom tags (produced by the first page producer) contained within can again be expanded by the second page producer's OnHTMLTag event handler into more custom tags where necessary. We can then assign the second page producer's Content to our third page producer's HTMLDoc property where the final content is produced and assigned to the Response object's Response property. The following code shows how to chain page producers:

```
PageProducer2.HTMLDoc.Add(
  PageProducer1.Content);
PageProducer3.HTMLDoc.Add(
  PageProducer2.Content);
Response.Content :=
  PageProducer3.Content;
```

Obviously this can only work with TPageProducers, the other page producer classes don't have an assignable content property like HTMLDoc. However, we can embed custom tags in the TDataSetTableProducer and TQueryTableProducer object's content by embedding the desired tag in the CellData parameter of the OnFormatCell event handler. Then it's a simple matter of assigning the dataset page producer's Content property to a PageProducer's HTMLDoc property. The PPCHAIN demo application (whose output is shown in Figure 1) called from the demo web page GETCUST.HTM demonstrates the use of chained page producers.

HTML-transparent tags can also contain any number of name=value parameters which follow the tag name, for example:

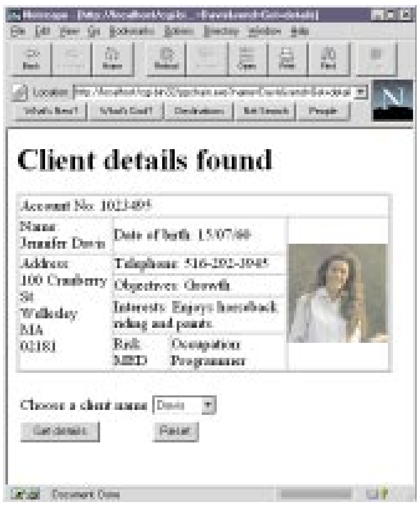
```
MYTAG ParamOne=AValue
  ParamTwo=AnotherValue
```

Notice that the parameters are delimited by a white space character which shows up a serious limitation: parameter values cannot include white spaces and hence the following tag will fail:

```
ADDRESS Street=95 The Boulevard
  Town=New Town County=Surrey
```

Fortunately there is a simple way around this: convert all spaces to plus (+) characters as is done with URL-encoding. The HTTPDecode and

➤ *Figure 1*



Client details found

Account No: 1023495		
Name Jennifer Davis	Date of birth 15/07/60	
Address 100 Cranberry St Wellesley MA 02181	Telephone 516-292-3945	
	Objectives: Growth	
	Interests: Enjoys horseback riding and paints	
	Risk MED / Occupation Programmer	

Choose a client name Davis

➤ *Listing 1*

```
if CompareText(TagString, 'ADDRESS') = 0 then
  ReplaceText := format('<#ADDRESSLINES ADDR1=%s CITY=%s STATE=%s ZIP=%s>',
    [HTTPEncode(FieldByName('ADDRESS_1').AsString),
     HTTPEncode(FieldByName('CITY').AsString),
     HTTPEncode(FieldByName('STATE').AsString),
     FieldByName('ZIP').AsString])
```

HTTPEncode functions provided in the HTTPApp unit make this very simple and are used in the PPCHAIN demo as shown in the Listing 1 code snippet which builds a custom tag with parameters whose values are taken from the CLIENT table in the DBDEMOS database. The HTTPEncode and HTTPDecode functions take the string to encode or decode as their only parameter.

The nice thing about the above workaround is that there's no need to call HTTPDecode to remove the + characters from the tag parameters when handling the OnHTMLTag event, this will be done by the ExtractHeaderFields procedure (defined in HTTPAPP.PAS) called by the ExtractHTTPFields procedure (also defined in that unit) which is called by the page producer's protected ContentFromStream method which, among other things, extracts any HTML-transparent tags and passes them to the protected HandleTag method from where the OnHTMLTag event handler is called to resolve any custom tags. ContentFromStream is called by the page producer's Content function.

The HTTPAPP unit does contain some other useful utility routines worth checking out.

Creating your own page producer component is pretty straightforward. A basic page producer whose content cannot be directly specified or modified by the component user should be descended from TCustomContentProducer and it's Content method overridden. Whilst such simple page producers may be limited in use, they can be very useful. I've created a very simple TFieldsProducer component which just displays the request fields (if any) for the WebDispatcher's Request object: just drop the component on to your WebModule and, in a TWebActionItem's OnAction handler, assign it's Content method result to the Request's Content property. Try it for yourself. You might find it useful for debugging your own web applications. You might want to go much further and extend TFieldsProducer to display all properties of the Request object, by then you'd probably want to call the component something like TRequestPropsProducer!

Custom page producer components are potentially very powerful: the obvious candidate for encapsulation as a custom page producer is a page hit counter which could display the hit count as text or graphical digits as commonly seen in commercial web pages. Another potential custom page producer is one which returns HTML formatted real-time information from some remote data-source accessed via the PC's communications port for instance.

Saving State

Some web applications need to interact with multiple web pages or web sessions, but because HTTP is a "stateless" protocol, a web application cannot access the request data of any previous requests. In other words, state is not maintained between web pages. I'm sure at some time or other during your quality web-surfing time you've encountered multi-page web applications (such as virtual shopping malls, web search engines etc) which somehow manage to pass request data (ie save state information) between web pages and even web sessions!

So how's it done? Bob Swart discussed a few approaches to state saving in September's issue, here I'll discuss the various approaches with a detailed comparison.

There are three approaches to state saving, ranging from the simple and reliable to the clever but not 100% reliable. The former approach is to embed form field values in the URL of the document returned by the application, as is done by the STATE1 demo application called by MARKETING.HTM to process it's FULLNAME form field. The output of STATE1 is shown in Listing 2.

In this case the name "Jaimie" was typed in the edit box of MARKETING.HTM and is included in each of the two hyperlinks which execute the THANKYOU demo web application. The main problem with this approach is that in situations where many variables are being passed between many web pages, the resulting URL can become too long for the browser to handle and information is lost. Not only that, confidential form data such as passwords will be visible within the URL, not very clever. But, for state saving, this method is by far the most reliable and universal. This approach to state saving uses the GET request method with all its usual limitations.

To overcome these limitations we can use hidden form fields which can be POSTed to the web application and so aren't embedded within the URL. To use this approach we must use submit pushbuttons instead of hyperlinks as used in STATE1. The STATE2 demo web application which produces the output in Listing 3.

The hidden HTML input type isn't displayed by the browser, it's effectively a hidden edit box hence the name. Hidden form fields are treated just like normal form fields but they have one major drawback: persistence, or rather, a lack of it because a hidden field's value is lost as soon as another page is called, which makes hidden fields less useful for saving state.

Which leads us to cookies, the clever but not 100% reliable approach: let's see why.

State-Saving Cookies

HTTP Cookies (or just cookies) are name=value pairs sent to and stored by the client. Cookies are provided to make state saving between web pages and web sessions easier and more reliable. Support for cookies became available with earlier versions of Netscape Navigator and are also supported by Internet Explorer 3.0x: given that both browsers are widely used it's fairly safe to say that you won't be excluding a large proportion of visitors from your web page if it makes use of cookies. Cookies are sent using the Set-Cookie response header field of the form:

```
Set-Cookie: NAME=value; EXPIRES
  Dayname, DD-MM-YY HH:MM:SS
  GMT; PATH=path; DOMAIN=
  domain; secure
```

```
<HTML><HEAD><TITLE>Maintaining state using GET request method</TITLE></HEAD>
<BODY><H2>Thanks for your details Jaimie</H2>
<H3>Would you like to receive marketing information from other subsidiaries of
  Acme Consulants Ltd?<BR><BR>
<A HREF="http://localhost/scripts/thankyou.dll?fullname=Jaimie&include=Yes">
  Yes I certainly would</A><BR>
<A HREF="http://localhost/scripts/thankyou.dll?fullname=Jaimie&include=No">
  No I definitely wouldn't</A>
</H3></BODY></HTML>
```

➤ *Listing 2*

```
<FORM ACTION="http://localhost/scripts/marketing.dll" METHOD="POST">
<INPUT TYPE="SUBMIT" NAME="include" VALUE="Yes">
<INPUT TYPE="SUBMIT" NAME="include" VALUE="No">
<INPUT TYPE="HIDDEN" NAME="fullname" VALUE="Jaimie Sach">
</FORM>
```

➤ *Listing 3*

```
Cookies := TStringList.Create;
Cookies.Add('Delphi=Cool');
Cookies.Add('JBuilder=Wait and see');
// set cookies which will be sent back only to the current domain and path
// their expiry date is 7 days from today and will be sent over a non-secure
// HTTP connection
Response.SetCookieField(Cookies, '', '', Now + 7, False);
Cookies.Free;
```

➤ *Listing 4*

When the browser accesses the domain which sent the cookie, it sends the cookie back to the server which passes it to the CGI program. Cookies can have optional attributes to further refine their usefulness: the EXPIRES attribute specifies a date, even a past date, on which the cookie will expire. By default cookies expire as soon as the web session ends but by setting a future date the cookie will persist beyond the browser session; by setting a past expiry date the cookie is deleted. Note that the time portion of the EXPIRES attribute is expressed in terms of GMT, so be careful if your web application resides on a server located outside that time zone.

The DOMAIN attribute specifies the domain to which the cookie will be sent, by default the cookie is sent to the domain name of the server which sent the cookie, but you can only specify a value within the domain of the server. So if your web application resides on www.delphimag.com/cgi-bin your cookie can have a domain attribute of delphimag.com/cgi-bin/other or just delphimag.com but not www.borland.com. This makes perfect sense: if it were possible for you to set cookies to be sent to another domain, you could write applications which would spam any web site with unwanted cookies. The PATH attribute specifies a substring of the path from the URL and defaults to the path of the web application, a cookie with a PATH of / will be sent to the server whenever the specified domain is accessed. The secure attribute specifies that the cookie will only be sent to the server over a secure HTTP connection. Note that cookie attributes are separated by a semicolon and a space.

Both Navigator 3.0 and IE 3.0x provide the security option of accepting or rejecting a sent cookie: the full cookie attributes are displayed in a dialog, useful for testing your cookie-bearing web applications.

To send a cookie in your Delphi application use the SendCookieField method of TWebResponse shown in Listing 4.

Cookies can be very useful but are not universally supported by all browsers, though I'd bet that the majority of web surfers are using cookie-friendly browsers. The other drawback to using cookies is that they're invasive: because they're stored by the browser they take up hard disk space which can amount to quite a lot, especially if many cookie-bearing web sites are accessed. Furthermore, both Netscape 3.0 and Internet Explorer 3.0x have a security option to warn you that a cookie is about to be sent to the browser, this cookie can be rejected by the user in which case your state saving strategy will be broken. Also there is a maximum number of cookies that the browser can handle: Navigator can handle around 20 cookies per domain, as well as the issue of the maximum length of the Cookies request header field. And finally it's possible that the surfer may delete or modify any stored cookies using any one of the cookie management utilities currently available! Oh well.

The DIARY demo application demonstrates the use of cookies for maintaining state. Running DIARY for the first time will present you with a web page into which you can enter your name. After submitting the form two cookies, your unique ID and name, are sent by DIARY to be stored by the browser. Each time DIARY is accessed, the cookies (which identify you) are sent to the application which allows it to correctly identify you. Diary entries can be added and retrieved and are stored in a table named WDIARY residing on the web server in the DBDEMOS database.

Besides passing form data between web pages and web sessions, cookies are also useful for storing information unique to the web surfer such as a username for accessing a private web page, customisation settings for a web application and anything else you care to think of, provided the cookie data is fairly short!

I want to conclude this section on cookies by laying to rest the serious misconception that cookies are capable of wreaking all sorts of havoc with your PC. Cookies are just data passed between the browser and server, they cannot execute any code on the browser or server so just ignore the myths of cookies sending information back to the web server about applications on your hard disk or the one about cookies reformatting hard disks or deleting the registry: the only thing a cookie can do is send information which you explicitly consented to accept in the first place, assuming you had your browser's cookie confirmation security option enabled.

Whilst we're on the topic of web security, it's worth mentioning that ActiveX controls (supported by IE 3.0x and above) embedded in web pages have access to the Win32 API and therefore can certainly erase files or reformat your hard disk, if that's what the ActiveX control's author intended!

Borland have published a web document (www.borland.com/security.htm) discussing their use of cookies as well as dispelling their mythical dangers.

URL Redirection

Web applications can redirect the server to fetch its response content, usually a web page, from a specified location. This can be useful where the web application must display a custom web page to each individual user or group of users, but without the need for the application to generate the customised web page.

Redirection can be achieved in one of three ways. The first, most straightforward, method is to call the response object's `SendRedirect` method which takes a URL string parameter. Do this if you just want to redirect the request without doing anything clever.

The other methods involve setting the response instance's `StatusCode` and `Location` properties. The status code 301 indicates that the requested web page has been permanently moved to the location specified by the `Location` property. 302 indicates that the web page has been temporarily moved to the specified location. The way 301 and 302 responses are handled are browser-dependent: Navigator 3.0 simply displays the web page specified by `TWebResponse.Location` whereas Navigator 2.0 displays a page saying that the requested document has been permanently or temporarily moved and presents a link which points to the new location.

However to make redirection work with status codes requires that the response header contains the Location field as well as the 301 or 302 status code. To add the location header field to the response use the `Response` object's `SetCustomHeader` method, for example:

```
Response.StatusCode := 301;
Response.Location :=
  'docs\somewebpage.html';
Response.SetCustomHeader(
  'Location: ' +
  Response.Location);
```

If you're sharp you might wonder why I bothered setting `Response.Location`? Well there's no real reason: I could have just specified the actual URL as part of the string passed to `SetCustomHeader` and saved a line of code. Actually, the `Location` property is pretty much redundant unless you modify the `SendResponse` method to include it as part of the response header but I've elected to use the less invasive `SetCustomHeader` instead. This method is useful for adding response header items not yet supported by the `TWebResponse.SendReponse` method.

By the way, you shouldn't send any content as part of the response when specifying the `Location` header field.

Client-Side Form Validation

Form validation can use up a lot of network bandwidth. Consider this typical scenario. The browser submits the form to the server/web application, the web application validates the field values and sends back a response content containing an error message if the validation failed. The surfer must then go back to the form, correct the invalid field and re-submit the form. Over a slow HTTP connection this can be very time-consuming and possibly annoying for the web surfer. Ideally the form would be validated before submission to the server: if you're using the latest browsers from Netscape or Microsoft you can do just that. Both Navigator 2.0 (and above) and Internet Explorer 3.0x support HTML scripting languages.

The JavaScript scripting language is an object-oriented (not just object-based) derivative of Java and is supported by Navigator. Visual Basic Script (VBScript) is a stripped-down version of VB 4.0 supported by Internet Explorer which also supports JavaScript but calls it JScript for reasons best known to Microsoft. Navigator can also handle VBScript via a plug-in component.

The interpreted script code is contained within the `<SCRIPT LANGUAGE> </SCRIPT>` tags. Both languages are quite powerful: Netscape use JavaScript extensively in their FastTrack web server's remote configuration management tools to good effect; Borland's IntraBuilder uses both client-side and server-side JavaScript to build web applications. It's no surprise that VBScript is used in Microsoft's Visual InterDev web development tool.

The demo web page VALIDATE.HTM uses JavaScript to validate its form fields before allowing the request to be sent to the server. Take a look at the JavaScript source to see how it works: it's pretty straightforward and can be used as boiler-plate code for implementing validation in your own web forms.

Though I've barely touched on the details of HTML scripting, it's an increasingly important skill for the web developer to use for building interactive web pages and to add more functionality and polish to a web site. The example I've presented is the most obvious application of HTML scripting used in conjunction with a web application but much more can be achieved with JavaScript. If you're serious about web development then you'd do well to learn either or both scripting languages. My personal preference is JavaScript, as it is far more powerful than VBScript.

That's all I'll say about HTML scripting, there are plenty of books available on both scripting languages as well as language references on both Netscape and Microsoft web sites. Microsoft Developer Network (MSDN) members will find both JScript and

VBScript language tutorials and reference guides on their latest CD. There's also a paper comparing JavaScript and VBScript at www.centaur.com.

Before closing this section on forms validation, a note about empty form fields: they're ignored and don't get sent as part of the request. Generally this doesn't matter as the `Values` property of the `QueryFields` or `ContentFields` `TWebRequest` properties will return a blank string if the specified value name doesn't exist, but if your application handles a number of different HTML forms and tries to identify a particular form by the form field names received in the request then it will fail if any of the fields are submitted blank.

The Server APIs

Delphi 3.0 server API-based web applications support the two main server APIs: ISAPI and NSAPI both of which are handled by the `TISAPIApplication` class. But how? After all ISAPI and NSAPI are different server APIs so how can one set of classes handle both? Well the support for NSAPI is a rather clever cheat because what actually happens is that NSAPI calls are handled by a DLL called ISAPITER which manages a cache of ISAPI DLLs, `TISAPIApplications`, to handle NSAPI calls. ISAPITER simply delegates NSAPI function calls to an internal `ISAPISession` instance which translates the NSAPI call and passes control to the target ISAPI DLL (from a list of previously loaded or cached DLLs) which handles the request via the usual ISAPI entry points. If the target DLL isn't found in the cache then it is loaded, added to the cache and then called to handle the request. Borland refer to ISAPITER as their "NSAPI bridge" technology which in theory could be used to allow any ISAPI application to be used with an NSAPI server.

Configuring NSAPI

Borland describe how to configure Netscape servers to use ISAPITER (see page 34-23 of the *Developer's Guide*) but manage to omit one crucial step: you must insert the line:

```
NameTrans from="/scripts/*"
  fn="assign-name" name="isapi"
```

in the `<Object name=default>` section of OBJ.CONF which will enable ISAPITER to handle all requests to the path scripts/SomeISA.DLL. If that fails then you'll need to explicitly load the configuration file from within the browser-based server administration utility.

Configuring an ISAPI server, any of Microsoft's really, is a lot less work and just requires you to map a domain-relative virtual directory (usually /scripts) to the local directory containing the ISA DLLs. But so much for configuring ISAPI servers, let's take a brief look at how ISAPI/NSAPI requests are handled by a `TISAPIApplication`.

Inside ISAPI Applications

When the server receives a client request, it spawns a thread to handle that request which is passed to the `TISAPIApplication` instance, which in turn may create a new instance, or reactivate a cached instance, of the `WebModule` within the context of the request's thread. If the `Application.CacheConnections` property is set to `True` any new `WebModule` instance is cached for later re-use by another request thread so it's possible that a request might not create a new `WebModule` instance, new instances are created only when all cached `WebModules` are busy. Each `WebModule` is created or reactivated within the context of the calling thread, and is made thread-safe by the use of critical section thread synchronisation objects. This saves us the worry of thread conflicts in our ISAPI applications.

A `TISAPIApplication` can handle a maximum number of active `WebModules` (ie requests) as determined by the `Application` object's `MaxConnections` property, so if you find your ISAPI application running out of `WebModule` instances increase `MaxConnections` appropriately. The lower the value of `MaxConnections` the less memory the web application will use overall but obviously this must be balanced with the number of active requests which must be handled.

Although ISAPI has performance advantages over good old CGI it can be a pain to develop with. I don't mean that it's difficult, Delphi 3.0 makes it fairly easy, but when you want to copy the latest version of your application over the existing one you may be prevented from doing so because if the server has already loaded a previous version of your DLL it won't let go of it unless you stop the server. Fortunately there's a solution, for Microsoft servers anyway, start up Regedit and change the value of the key

```
HKEY_LOCAL_MACHINE\System\
  CurrentControlSet\Services\
  W3Svc\Parameters\
  CacheExtensions
```

from 01 to 00, this will cause the server to load and unload the DLL after handling each request so there's no need to shut down your development server each time you want to test the latest version of your pet project. Unfortunately I know of no way to achieve the same thing with NSAPI servers. One word of warning: make sure `CacheExensions` is set to 01 on your live web server otherwise performance will be seriously degraded.

Although the `WebModule` makes writing ISAPI applications relatively easy, some care is still required especially with writing database ISAPI applications. ISAPI database web applications occasionally run into problems with the BDE and, on a number of occasions, I've had "illegal BDE reentry" faults. The BDE is non reentrant: if during the internal execution of a BDE API function another BDE function is called the BDE's internal call stack will become corrupted and cause the program to crash. Re-entry can occur in multi-threaded applications, which is where the BDE Session comes in. A thread which uses the BDE must first open a new IDAPI session which will isolate or block all internal BDE operations between different threads. Most importantly a session opened by a thread should be used exclusively by that thread only: before the thread terminates it should close the session so that no other thread can use it.

A WebModule used by an ISAPI application should have a TSession component with it's AutoSessionName property set True so that a new session name is automatically generated and used by all datasets in the WebModule: never rely on the default TSession or you'll run into problems. With AutoSessionName set, the TSession in each WebModule instance created by a server thread spawned to handle a request will have a unique session name. So far so good, but if the Application's CacheConnections property is set, a cached WebModule will be re-used to handle the request in which case the session opened by a previous thread will still be active and used in the context of another thread which didn't open that session, this situation seems to cause the intermittent illegal BDE re-entry faults.

There are two workarounds: either set CacheConnections to False or in the WebDispatcher's BeforeDispatch and AfterDispatch events, open and close the session which will close dependent datasets: this will ensure that a new session is created when the WebModule is re-used and BeforeDispatch opens the session and it's datasets, but at the cost of the complete benefits of using cached WebModules. A definite case of choosing the lesser of two evils?

Of course you'll never encounter such problems with CGI/WinCGI web applications, for which the CacheConnections and MaxConnections properties of the Application object are irrelevant.

Converting Between CGI And ISAPI

Converting an ISAPI/NSAPI application to CGI/WinCGI is a simple matter of editing the project source so that an executable (program) rather than a DLL (library) is produced. Listing 5 shows the differences between the two types of project source. If you don't want the hassle of editing the project source you can just run the Web Server Application Wizard, choose the application type you want and then replace the default WebModule with your own. A neat solution would be to write an IDE Expert to make the necessary alterations to the project source, now there's a job for someone...

```
program CGIApp;
{$APPTYPE CONSOLE}
uses
  HTTPApp, CGIApp,
  webmod in 'webmod.pas' {WebModule1: TWebModule};
  {$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TWebModule1, WebModule1);
  Application.Run;
end.

library ISAPIApp;
uses
  HTTPApp, ISAPIApp,
  webmod in 'webmod.pas' {WebModule1: TWebModule};
{$R *.RES}
exports
  GetExtensionVersion,
  HttpExtensionProc,
  TerminateExtension;
begin
  Application.Initialize;
  Application.CreateForm(TWebModule1, WebModule1);
  Application.Run;
end
```

➤ *Listing 5*

Get Out The Bug Spray!

The CGI/WinCGI classes have a few bugs which are serious enough to prevent CGI and WinCGI apps from working on the web servers that I've used, but seeing as most Delphi 3.0 web applications will run on either Netscape or Microsoft web servers using server APIs you might not be too worried about such bugs! Okay so CGI and WinCGI are less important than ISAPI and NSAPI in the Windows web server world, but there are times when CGI is the better option, especially when your ISAPI DLL keeps pulling down your web server as a result of elusive memory leaks and/or fatal crashes. It's also handier to develop and test your web server applications using CGI or WinCGI, as you're saved the hassle of stopping the web server in order to unload the previous version of your ISAPI DLL each time you want to test changes to your latest creation (though there's a way around this for Microsoft servers as we've seen). And finally, not all Windows-based web servers support ISAPI or NSAPI so we can't just forget about CGI yet awhile.

So for the benefit of the (probable) few, let's get out the bug spray and eradicate those bugs. The first bug (?) applies to O'Reilly's Website Server 1.1 and causes CGI applications to generate an *Unknown status reply from server: 0!* error dialog before displaying the response received from the server. Examination of the response header sent by the server shows the first line to read HTTP/1.0 OK instead of the correct HTTP/1.0 200 OK. With Netscape FastTrack Server 2.0 there's no such problem.

Anyway for WebSite users the fix is to add the following lines to CGIAPP.PAS:

```
{$IFDEF WEBSITE}
AddHeaderItem(StatusString,
  'HTTP/1.0 %s'#13#10);
{$ENDIF}
```

before the line:

```
AddHeaderItem(StatusString,
  'Status: %s'#13#10);
```

which cures the problem completely (assuming that you have defined the WEBSITE conditional define). It also fixes the problems of cookies with past expiry dates not deleting themselves and of StatusCode and Reason response properties being ignored. This fix applies only to Website 1.1 which seems to imply that the problem is with that particular server.

The second bug causes TCGIResponse.SendRedirect to fail, again the fix is simple. In the implementation of SendRedirect, append #13#10#13#10 to the format string, this ensures that there's a blank line after the response header as required by HTTP. This bug affects any web server.

The next few bugs prevent WinCGI from handling POST requests and from creating the response output file when used with non-Microsoft servers. The first fix is to add the following line to TWinCGIRequest's constructor:

```
if ContentFile = " " then
```

before the line:

```
FClientData := TFileStream.Create(
  ContentFile, fmOpenRead  or
  fmShareDenyNone);
```

which fixes the problem of the application failing when the request method is POST.

The second fix requires a change to the following line in the same constructor:

```
FServerData := TFileStream.Create(
  OutputFile, fmOpenWrite  or
  fmShareDenyNone);
```

to read:

```
FServerData := TFileStream.Create(
  OutputFile, fmCreate  or
  fmOpenWrite or
  fmShareDenyNone);
```

which handles the situation where the web server doesn't create the response output file but leaves the web application to do so, as is the case with the WebSite and Netscape servers.

The final bug prevents cookies from being retrieved by TWinCGIRequest and is fixed by simply replacing the following line in TWinCGIRequest.GetStringVariable:

```
21..24, 26..28:
```

with:

```
21..24, 26,28:
```

This causes the Cookie key value to be correctly retrieved from the [Extra Headers] section of the request INI file rather than from the [CGI] section where the key doesn't exist.

And that's it. Before making these changes to CGIAPP.PAS I'd recommend that you copy the source from SOURCE\INTERNET to another directory called, if you like, SOURCE\CGIFIXED.

Debugging Web Applications

The most convenient way to debug a web application is to make it a WinCGI application. Most WinCGI-capable web servers provide a facility to prevent the server from deleting the request data INI file when the WinCGI application terminates, but if not there's nothing to stop you writing a routine in your WinCGI application to copy the request INI file for later use debugging your apps. By specifying the saved INI file path as a Run parameter you can just run and debug your application from within the IDE like any other.

Both the above approaches will work fine provided any bugs are contained within your application's code and not within the component library's code, obviously any bugs in the ISAPI/NSAPI component and class libraries can't be tracked down using a WinCGI application as the testbed and vice versa!

So You Wanna Play The Web Master?

I'm hoping that after reading all this you're just itching to try writing your own web applications (if you haven't already done so), but perhaps don't have an intranet to play with, let alone a web server to dish up your web pages and applications. Well don't despair because you don't have to have an intranet, a single PC will do, and there are some free web servers available out there too.

If you want to write CGI and WinCGI web applications you can install O'Reilly's WebSite 1.1 Server, which I've seen free on magazine cover CDs, is available at www.ora.com and is also included with the book *Building Your Own Web Site*. It's published by O'Reilly & Associates, who also publish some excellent books for internet developers. A trial version of WebSite 2.0 Professional, which adds loads of new features, is available for download from www.ora.com.

For your ISAPI needs there's Microsoft's Personal Web Server (PWS) for Windows 95, or Internet Information Server (IIS) for Windows NT, both available from www.microsoft.com. For NSAPI, trial versions of Netscape's web servers can be downloaded from home.netscape.com. Netscape FastTrack 2.0 runs under Win95 and supports both CGI/WinCGI and NSAPI. You can also add support for CGI/WinCGI to Microsoft's web servers.

Once you've successfully installed the web server you should be able to access it from the same machine using the URL http://127.0.0.1 (or http://localhost with WebSite, or the hostname configured with Netscape Server) from your browser, but be sure to use Navigator as your browser as I've found that IE 3.0x sometimes behaves strangely when used in this way. As part of the server setup/configuration it's usually possible to configure the hostname you wish to use to identify your web server.

If you find you can't connect to your local web server and you're not on a network, then install dial-up networking and bind TCP/IP to the dial-up adapter using the IP address 127.0.0.1 (for example) with a subnet mask of 255.0.0.0, cancel the error dialog about the IP address being invalid, and then exit Control Panel. You'll then be prompted to insert your Windows 95 (or NT) CD-ROM so that the necessary drivers can be installed, followed by a system restart. After this you should then be able to start your local web server from your Netscape browser.

You may be wondering about the minimum hardware required to host a web server. Well I've managed to run WebSite 1.1 and Personal Web Server 95 on a 486/66 with 32MB RAM with no problems, and all the research and testing (using WebSite, PWS 95 and Netscape FastTrack 2.0) for this

Server Directory	Maps To
/docs/	Directory containing sample web pages
/cgi-bin32/	Directory containing CGI applications
/cgi-win32/	Directory containing WinCGI applications
/scripts/	Directory containing ISAPI/NSAPI application DLLs

➤ *Table 1*

article has been done on a Pentium 120 notebook with 32Mb RAM.

To run the demos presented here you'll need to configure the server directories listed in Table 1 on the web server to map to the local directories shown.

The server administration facilities of all the web servers I've mentioned are straightforward to use for configuring virtual directories: both Microsoft and Netscape servers use remote browser-based administration front-ends. All the demo applications are ISAPI/NSAPI applications which you can use with any Microsoft or Netscape server. If you want to get up and running quickly under Windows 95 I recommend running the demo applications under Personal Web Server: it's free and the download file size is relatively small.

Conclusion

Whilst the new web application classes and components save a lot of effort there are times when it might be better to write your web application from scratch, foregoing the luxury of the VCL and using plain old "back to basics" Pascal.

For simple web applications the overhead of using a WebModule can be significant: 200Kb against 36Kb for a very trivial "hello world" WebModule application against the equivalent non-WebModule application. If you're writing ISAPI/NSAPI applications it's better to use WebModules, but simple CGI/WinCGI applications can be made significantly smaller if WebModules are not used, at the possible expense of having to write more code. A case of the classic trade-off between development time and code size: the choice is yours.

If you do decide to write your simple CGI web applications without using WebModules then check out previous articles by Bob Swart and Steve Troxell. You can also check out my paper (plug, plug...) on writing web applications with Delphi 2.0 on the Borland Developer's Conference 1997 CD-ROM.

John O'Connell is a freelance developer and consultant specialising in Windows and Internet software development. He can be emailed at john.oconnell@btinternet.com when not travelling around the world, in which case he picks up email at john_o_connell@hotmail.com