Under Construction: Delphi 3 Web Modules, Part 1

by Bob Swart

This month, we'll explore a new feature included only with the Client/Server version of Delphi 3: web modules. If you don't have the Client/Server version, this article may help you decide whether to purchase it!

Web modules come with a number of new components (found on the `Internet` tab of the Component Palette) and a new Wizard for starting a Web Module project (which can be found in the Repository after a `File|New`).

If we select the Web Server Application option the Wizard asks which type of Web Server application we'd like to build (Figure 1). ISAPI/NSAPI DLLs have the common advantage that these processes on the Web Server typically only have to be loaded once and can remain resident after the first load, so they eliminate the time-intensive loading/unloading we get when using CGI and WinCGI web applications. However, since the internal logic isn't much different (certainly not for the Delphi 3 web module programmer), I decided to build a WinCGI web application this time, so we can use the Intra-Bob CGI Debugger version 2.01 (available on my new website at www.drbob42.com) to test it on our local machine without the need for a local web server.

Web server applications extend the functionality and capability of existing web servers. The application receives HTTP request messages from the web server, performs any actions requested in those messages and formulates responses that it passes back to the web server. Any operation we can perform with a (non-visual) Delphi application can be incorporated into a web server application.

Table 1 shows the four types of web server applications and the corresponding objects. Each type of application uses a type-specific descendant of `TWebApplication`, `TWebRequest` and `TWebResponse`.

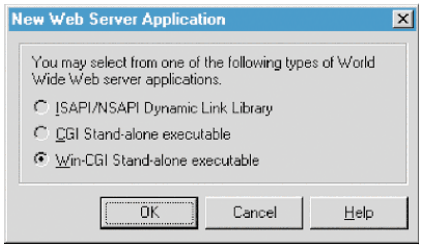
Application Type	Application Object	Request Object	Response Object
Microsoft Server DLL (ISAPI)	TISAPIApplication	TISAPIRequest	TISAPIResponse
Netscape Server DLL (NSAPI)	TISAPIApplication	TISAPIRequest	TISAPIResponse
Console CGI Application	TCGIApplication	TCGIRequest	TCGIResponse
Windows CGI Application	TCGIApplication	TWinCGIRequest	TWinCGIResponse

➤ *Table 1: Web application types*

For an ISAPI or NSAPI application client request information is passed to the DLL as a structure and evaluated by `TISAPIApplication`, which creates the dispatcher, `TISAPIRequest` and `TISAPIResponse` objects.

A CGI standalone application is a console application that receives client request information on standard input and passes the results back to the server on standard output. This data is evaluated by `TCGIApplication`, which creates the dispatcher, `TCGIRequest` and `TCGIResponse` objects.

A Win-CGI standalone application is a Windows application that receives client request information from a configuration settings (INI) file written by the server and writes the results to a file that the server passes back to the client. The INI file is evaluated by `TCGIApplication`, which creates the dispatcher, `TWinCGIRequest` and `TWinCGIResponse` objects.



➤ *Figure 1*

```
program Project1;
{$APPTYPE GUI}
uses
  HTTPApp, CGIApp,
  Unit1 in 'Unit1.pas' {WebModule1:
TWebModule};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(
    TWebModule1, WebModule1);
  Application.Run;
end.
```

➤ *Listing 1*

WinCGI

After we click OK in the Web Module Wizard, we get a new project with a new empty web module (instead of the regular empty new form). A web module is just like a new form, however, since it's auto-created in the same way as a regular form. The main project source file contains code to prove that (Listing 1).

Note that this WinCGI project uses the `HTTPApp` and `CGIApp` units, whilst an ISAPI/NSAPI project generated this way will use the `HTTPApp` and `ISAPIApp` units. The `{$APPTYPE GUI}` further specifies that it's a WinCGI application, as opposed to a standard CGI application that specifies `{$APPTYPE CONSOLE}`. Of course, in both cases the

application is non-visual, but the CGIApp unit uses the APPTYPE compiler option to distinguish between standard CGI and WinCGI applications. But they offer a quick way to change a WinCGI application to a standard CGI application or *vice-versa*. We can even switch to an NSAPI/ISAPI application, by using the ISAPIApp unit instead of CGIApp. Other than that, the different internet protocols are practically invisible to the Delphi developer, so just pick the protocol you like most and join the rest of the article (you'll need to read the manual for tips on how to debug ISAPI/NSAPI applications).

A web module is actually more like a data module, in that we can only drop non-visual components on it. While we'd put data access components on data modules, for web modules we can also drop components from the Internet tab (Figure 2), but not the visual NetManage ActiveX controls of course.

Specifically, we can use one or more of the following new components (from left to right, skipping the first two components and the last eight from NetManage):

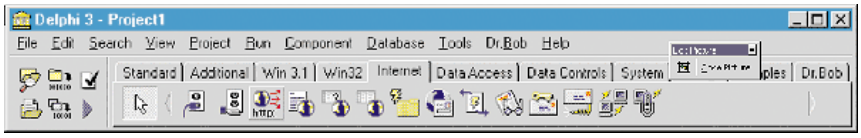
```
TWebDispatcher
TPageProducer
TQueryTableProducer
TDataSetTableProducer
```

TWebDispatcher

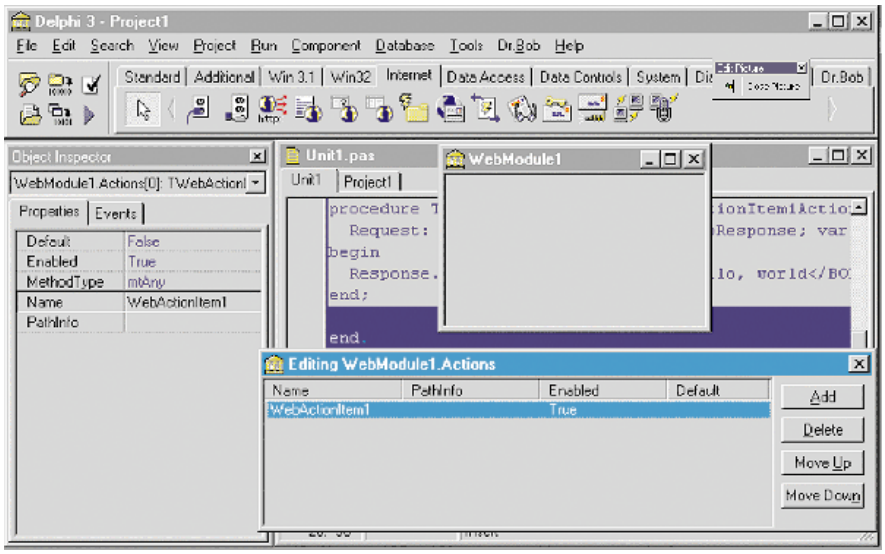
This component is already built into the web module, but we need one if we want to use an existing data module as a web module. We can't drop a TWebDispatcher component into a web module as there should be only one per module. As you would expect, this component is the "dispatcher" of actions and events.

The property Actions of type TWebActionItems contains a list of all actions that this web application can perform. Each action has a number of subproperties: Name, PathInfo, Enabled and Default.

The PathInfo is the action request that is sent to the web server and can be used to identify subtasks to be performed by this single web application (so we can actually merge several web actions into one web application, instead of having to write a separate web application for each action).



➤ *Figure 2*



➤ *Figure 3*

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
   Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '<HTML><BODY>Hello, world</BODY></HTML>';
end;
```

➤ *Listing 2*

If we click the Add button, the first action item will be created, named WebModule1.Actions[0] of type TWebActionItem, but with an alias of WebActionItem1 (Figure 3). I say alias because there's something going on behind the scenes here: the Object Inspector will use the WebModule1.Actions[0] name in the component combobox, while the actual name of the component will be WebActionItem1. The form definition in the code editor contains no reference to a WebActionItem component, so it should be clear by now that we're dealing with a sub-component of the web module, namely Actions[0], that for our convenience is named WebActionItem1.

Since it's the only WebAction item, it'll be the default one as well. To actually write code for this first WebAction item, we have to select the WebActionItem1 in the Web Actions property editor, which will make the Object Inspector focus on WebModule.Actions[0] as well. Then, go to the events page, double click on the event OnAction and write the code shown in Listing 2.

In this OnAction event handler, we can fill in the Response of our WebAction, which usually generates a dynamic HTML file. Response is of type TWebResponse, which is an abstract base class for all objects that represent HTTP messages sent in response to an HTTP request message.

Our web application automatically creates a TWebResponse object based on the TWebRequest object for an incoming HTTP request message. The TWebDispatcher for the application then passes the TWebResponse object to the TWebActionItem associated with the TWebRequest object, so that the response can be formulated. Of the

many properties of the TWebResponse, the Content (of type String) is the most important: here we can assign whatever value we need to return (eg a dynamic HTML page).

Normally, we would also need to specify the format of the dynamically generated output, like content-type: text/html, but this can be specified in another property of TWebResponse, namely ContentType, which has the value text/html by default. In our first *Hello, world* web application, we've only defined the dynamic HTML output page, to consist of a single line that should display Hello, world.

IntraBob v2.0

Since we've already written one line of code, let's see if we can get some immediate feedback and test our Delphi 3 WinCGI web application. A few issues back, we wrote IntraBob (the latest version 2.0 is available on my website at www.drbob42.com and is compatible with Delphi 2.01, C++Builder and Delphi 3), a CGI and WinCGI testing application. We can use that to test our project. For this, we need an HTML CGI test form, defined as follows:

```
<HTML>
<BODY>
<FORM ACTION="project1.exe"
   METHOD="POST">
<INPUT TYPE=SUBMIT>
</FORM>
</BODY>
</FORM>
```

If you know HTML you will notice immediately that this form will send no actual data to the web application, since there is no input type other than the Submit button available. That won't be a problem, however (or at least, it shouldn't be!). Loading this HTML form in IntraBob, we only need to specify in the CGI Options page that we're dealing with a WinCGI application (so the options will be written to project1.ini, which will be given as a command line argument to project1.exe). If we click on the Submit button, however, we don't get the expected result. Instead, an exception dialog pops up, showing exception EFOpenEror.

Apart from the reason why we get this exception dialog, it proves once again the value of the IntraBob local CGI tester: a pop-up exception message on a Web Server would not be seen by anyone (except the web master if he happens to be near) and would have the same effect as hanging the Web Application... Not the best way to keep your web space provider happy!

So, what did we do wrong? Well, it turns out that there are two ways to execute a WinCGI application. One is by only specifying the inifile as a command-line argument to the web application and trust the web application to read the inifile and obtain the "form literal" data and "output file" specification from it. This is the technique that IntraBob uses. However, an earlier way for WinCGI applications to execute is by supplying not one but three command line arguments: the second containing the name of the file containing the form data and the third containing the filename to write the output to. So, if we go to the command line and call project1.exe again, but this time with project1.ini as the first argument, any (or a non-existing) file as the second argument and output.htm as the third argument, things should work again. Well, close but no cigar, yet. We still get the same exception: cannot open file. Surely, it can't be the input file (it doesn't even need to read the input). But it turns out to be the output file that must already exist (so our web application only has to overwrite it). So, if we create an empty file output.htm, and call project1.exe project1.ini nul output.htm then things work fine and we indeed get the HTML result in output.htm that we would expect.

The fact that this is indeed a bug can be found in the TWinCGIRequest.Create constructor, which is defined (in CGIApp.pas) as shown in Listing 3.

We see that if the ContentFile and OutputFile are not specified on the command line they are obtained from the inifile. But we also see that the FServerData OutputFile is opened using the fmOpenWrite or fmShareDenyNone flags. Which means that if the output file doesn't exist it won't be created! This is usually a problem the first time we run the WinCGI application on a new web server (unless the output file is removed after each request). The fix is to change the fmOpenWrite to the fmCreate flag (on lines 410 and 507 in CGIApp.pas). The fmCreate will open the file in fmOpenWrite mode if it already exists, or just create it for writing if it doesn't.

Note that O'Reilley WebSite 2.0 works exactly as IntraBob and will show the exception (because the output file is not created), while Microsoft IIS and PWS create the output file in the temp directory beforehand, so they give no problem.

Hmmm, a lot of trial and error to get even a minimal WinCGI app up and running. But that's usually the biggest problem: getting started.

TPageProducer

So far, we only wrote one real line of code, to assign a dynamic HTML string to Response.Content in the WebModule1WebActionItem1Action event handler. However, instead of writing our own HTML code from scratch here, we can use a more RAD approach, by dropping a TPageProducer component into the

➤ *Listing 3*

```
constructor TWinCGIRequest.Create(IniFileName, ContentFile, OutputFile: string);
begin
  FIniFile := TIniFile.Create(IniFileName);
  if ContentFile = '' then
    ContentFile := FIniFile.ReadString('System', 'Content File', '');
  if OutputFile = '' then
    OutputFile := FIniFile.ReadString('System', 'Output File', '');
  FClientData := TFileStream.Create(ContentFile, fmOpenRead or fmShareDenyNone);
  FServerData := TFileStream.Create(OutputFile, fmOpenWrite or fmShareDenyNone);
  inherited Create;
end;
```

web module and use it to produce a set of HTML commands (optionally based on an input template).

We should also change the `WebModule1WebActionItem1Action` event handler as shown in Listing 4.

The `TPageProducer` has two helpful properties to either store or direct to a HTML template file. The `HTMLDoc` property contains a `StringList` that stores the HTML template directly in the component itself, while the `HTMLFile` property contains a filename to the HTML template. This HTML template is used by the `PageProducer` to generate the `Content` string that is assigned to the `Response.Content` string (and returned from the web server application).

So, we can now click on the `PageProducer` component on the web module, click the `HTMLDoc` property in the Object Inspector and enter the contents of the HTML template in the string list editor (Figure 4). Too bad it doesn't do HTML syntax highlighting yet, but it wouldn't be hard to write a dedicated property editor for the `HTMLDoc` `TStrings` property of the `TPageProducer` component.

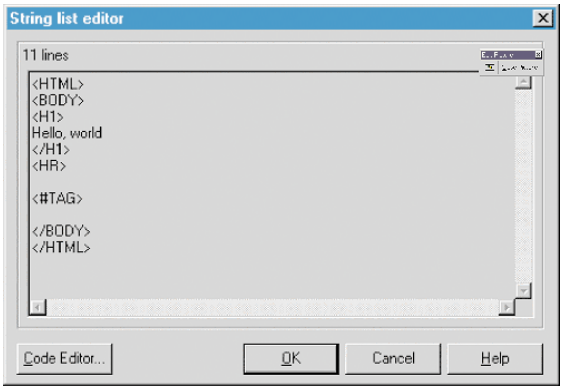
Each string is a sequence of one or more HTML commands or HTML-transparent tags. An HTML-transparent tag has the form:

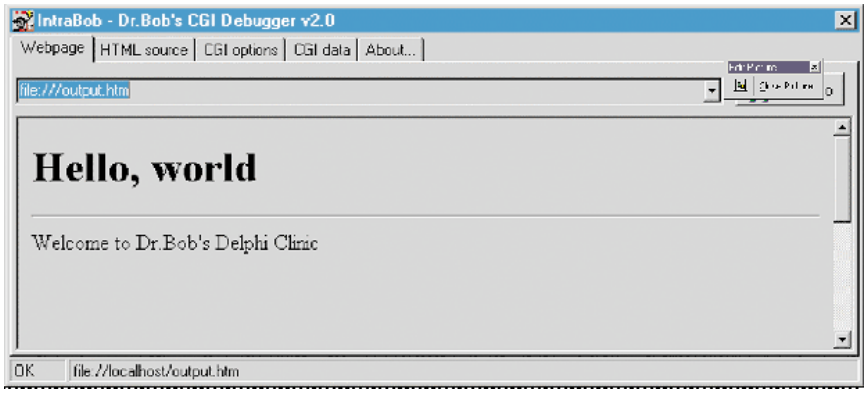
```
<#TagName Param1=Value1
 Param2=Value2 ...>
```

The `<#TagName` may not contain any spaces. The `TagName` identifies the HTML tag for the `PageProducer`, which converts the entire tag into a more meaningful HTML content based on the value of the tagname and the optional parameter values. This is a transparent tag, since normal HTML browsers don't recognise the `#TagName` construct and hence won't show anything.

The `Content` method converts `HTMLDoc` into a final HTML string by calling the `HandleTag` method to convert each HTML-transparent tag in `HTMLDoc`. So, after we've filled the `HTMLDoc` property, we should go to the events page of the Object Inspector to create an `OnHTMLTag` event handler for the `PageProducer`. Assuming there are more possible

➤ *Figure 4*





➤ *Figure 5*

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := PageProducer1.Content;
end;
```

➤ *Listing 4*

```
procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if TagString = 'TAG' then
    ReplaceText := 'Welcome to Dr.Bob''s Delphi Clinic'
  else
    ReplaceText := 'Error: Unknown tag offered...'
end;
```

➤ *Listing 5*

`TagNames` defined, we should check the `TagString` argument and for each specific `TagName` replace it with the required HTML. In this case, we want to replace the `<#TAG>` tag, which means we must look for a `TagString` with the name `TAG`. The optional Tag Parameters can be found in the `TagParams` argument. The initial `TagString` in combination with the `TagParams` can be used to define a wide range of tag values to replace.

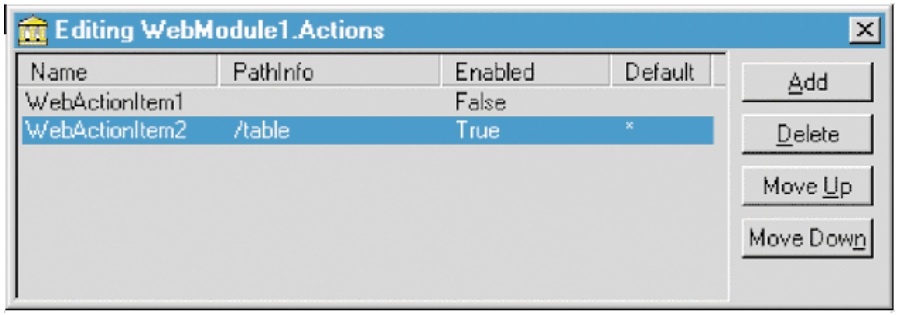
We can replace the Tag with any content we want and return that content in the `ReplaceText` string. See Listing 5.

If we test the WinCGI project we've written so far in IntraBob again, we get the expected result as shown in Figure 5.

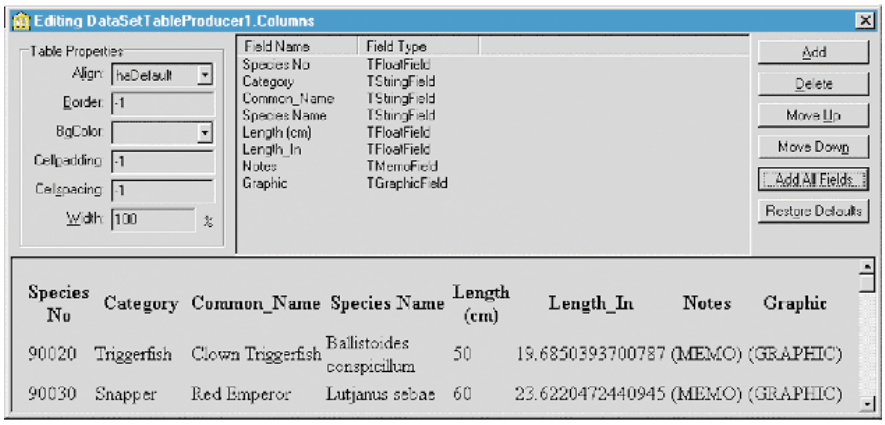
The `HTMLDoc` property in combination with the transparent Tags and the `PageProducer1HTMLTag` event handler gives us enough power to write a meaningful CGI, WinCGI or ISAPI/NSAPI web application. However, there's more. Much more...

Actions

For starters, we can put more than one action into a CGI web application. We can put a dynamic HTML



➤ *Figure 6*



➤ *Figure 7*

```
procedure TWebModule1.WebModule1WebActionItem2Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := DataSetTableProducer1.Content;
end;
```

➤ *Listing 6*

form in the main form and let it call another action. This can be done as follows, for example:

```
<FORM ACTION="project1.exe/table"
 METHOD="post">
```

Now, the `ACTION` doesn't consist of `project1.exe` anymore, but has a special action URL added to it: `/table`. The reason that this works is that the Web Server is able to parse a URI (Uniform Resource Identifier) into subparts. For example:

```
http://www.drbob42.com/cgi-bin/
 project1.exe/table?Name=DrBob
```

is parsed into the following parts:

```
protocol: http
host: www.drbob42.com
ScriptName: /cgi-bin/project1.exe
PathInfo: /table
Query: ?Name=DrBob
```

To define a new action for our web module, we need to click the web module `Actions` property again. In the `Editing WebModule1.Actions` dialog, click `Add` to get a `WebActionItem2`. Now, we need to set the `Enabled` property of the first action to `False` (since we have two `WebActions` and we only want to test the latest one) and we can specify the `PathInfo` of the second `WebActionItem`, for example as `/table` (Figure 6).

Note that I actually had to disable the first `WebActionItem` because IntraBob wasn't able to split the command-line into a URL, Query String, Logical Path and Physical Path. Hence, the `PathInfo` inside the WinCGI application was never set and the `WebActionItem1` was always fired from IntraBob (I'm working on supporting that as well). A real Web Server will be able to provide this information, either as environment variables (for standard CGI applications) or in the INI file for WinCGI applications.

Now that we have a second `WebActionItem`, it's also time to look at some alternatives to the plain HTML `TPageProducer` component: the `TDataSetTableProducer` and `TQueryTableProducer` components.

TDataSetTableProducer

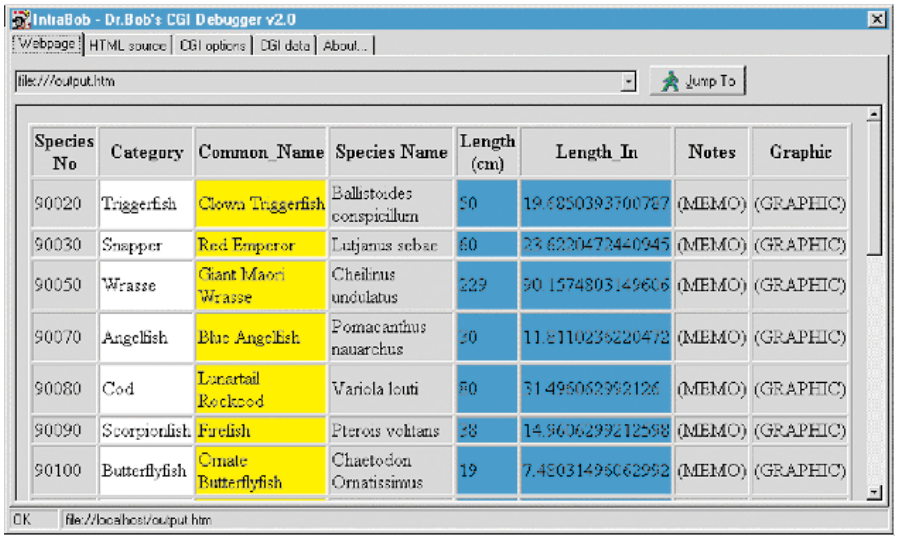
Usually, we don't want to put a simple HTML page on the web, but rather the contents of a database or the result of a dynamic query. In that case, we can use the more sophisticated `TDataSetTableProducer` and `TQueryTableProducer` (the `Table` part of their names refers to HTML table output, not to database tables). These two components can be used to produce nice looking HTML pages, consisting of records formatted in HTML-tables that look like `DBGrids` in the web browser.

In order to use the `DataSetTableProducer1` we must first connect to it from our second `WebActionItem`. This is done as shown in Listing 6.

The `DataSetTableProducer` must be connected to a `DataSource`, which means it's now also time to drop a `Table` (or `Query`) on the web module. Note that we don't need a connecting `DataSource` (which is only needed to connect visual data-aware controls to datasets). For this to work, the BDE must be installed on the Web Server, of course, including the demo databases and aliases.

For this example, we can drop a `TTable` on the web module, set the `DatabaseName` property of the `TTable` to `DBDEMOS`, the `TableName` property to `BIOLIFE.DB` and set the `DataSet` property of `TDataSetTableProducer` to `Table1` so we're ready to run.

In order to specify which columns are to be used in the `DataSetTableProducer` we need to set the `Active` property of the `TTable` component to `True`. Right after that, click the `Columns` property (type `THTMLTableColumns`) of the `DataSetTableProducer` component and we end up in a new property editor in which we can design the output the way we want it (Figure 7).



Species No	Category	Common_Name	Species Name	Length (cm)	Length_In	Notes	Graphic
90020	Triggerfish	Clown Triggerfish	Ballistoides conspicilium	50	19.685039370079	(MEMO)	(GRAPHIC)
90030	Snapper	Red Emperor	Lutjanus sebae	60	23.622047244094	(MEMO)	(GRAPHIC)
90050	Wrasse	Giant Maori Wrasse	Cheilinus undulatus	229	90.157480314961	(MEMO)	(GRAPHIC)
90070	Angelfish	Blue Angelfish	Pomacanthus nauarchus	30	11.811023622047	(MEMO)	(GRAPHIC)
90080	Cod	Lancetail Rockcod	Variola louti	80	31.496062992126	(MEMO)	(GRAPHIC)
90090	Scorpionfish	Firefish	Pterois volitans	38	14.960629921259	(MEMO)	(GRAPHIC)
90100	Butterflyfish	Ornate Butterflyfish	Chaetodon Ornatissimus	19	7.4803149606299	(MEMO)	(GRAPHIC)

➤ *Figure 8*

Most properties (like `Align` and `BgColor`) seem to work on the entire table instead of individual columns, at least at design time in the property editor, which is a pity). Actually, it turns out that if we want to specify formatting options for individual fields, we can click on any of these fields in the property editor, go back to the Object Inspector, and change them there (a bit unnatural: why can't we do it in the property editor?).

Anyway, after we close this property editor, we're ready to test the output of the `TDataSetTableProducer` (Figure 8).

Of course, this will look even better in a real web browser (rather than the NetManage HTML control that is the basis for IntraBob), and the colours used are probably not ergonomic, but you should have the idea by now. There are a lot more properties and events left unexplored at this time (such as the `Caption`, `Header`, `Footer`, `RowAttributes` and `TableAttributes` properties of the `TDataSetTableProducer`, and the `TQueryTableProducer` that I haven't covered at all). But I have to leave something for next time, right?

We've seen that Delphi 3 web modules offer a framework for CGI, WinCGI and ISAPI/NSAPI web applications that support multiple actions. The `WebModule` and `WebDispatcher` are responsible for dispatching the actions, while the `PageProducer`, `DataSetTableProducer` and `QueryTableProducer` can be used, with Tags and replacement contents, to create customisable dynamic HTML pages.

Next Time...

Next month, we'll use the information from this month's column to write a more complex CGI, WinCGI and ISAPI/NSAPI application (actually only one, but we'll see how to switch quickly from one protocol to another) that supports multiple states, tables, queries and more goodies.

Only for Delphi 3 Client/Server users, unfortunately, so I'm also working with Shoreline's Chad Z Hower on an article about Portcullis, the Internet Application Gateway from ShoreLine, showing how to write components compatible with IAG.

Bob Swart (aka Dr.Bob, visit www.drbob42.com) is a professional knowledge engineer technical consultant using Delphi and C++Builder, freelance technical author and co-author of *The Revolutionary Guide to Delphi 2*. Bob is now co-working on *Delphi Internet Solutions*, a new book about Delphi and the internet/intranet. In his spare time, Bob likes to watch videos of *Star Trek Voyager* and *Deep Space Nine* with his 3 year old son Erik Mark Pascal and his 8 month old daughter Natasha Louise Delphine.