Under Construction: Delphi 3 Web Modules, Part 2

by Bob Swart

This month, we'll continue our exploration of the Delphi 3 Client/Server Web Modules. We'll see how to write dynamic or parameterised queries, save state information and eat some cookies along the way.

Last time, we explored the Web Module Wizard, along with a few of the new Delphi 3 Client/Server internet components. Specifically, we examined the TWebDispatcher, TPageProducer and TDataSetTableProducer. This time, we'll take a closer look at the TQueryTableProducer and find out how we can send data back to the server instead of just generating a dynamic HTML page on the server.

TQueryTableProducer

At first sight, a TQueryTableProducer component looks just like a TDataSetTableProducer (after all, a Query is just another DataSet, right?). The difference can be found in the fact that the Query is a special DataSet, not just any DataSet, so the component can prepare for the fact that it's a Query. In fact, this brings us immediately to our second topic: sending input (data) back to the web server. Last time, we examined several ways to generate dynamic HTML pages using Web Modules, but that's only half of the story. Say we want to look again at the BIOLIFE table, but this time we only want to see the fish bigger than 42 centimetres. How would we do this? The obvious way is to use a query on the server, and connect the output to the TQueryTableProducer just like we did last time. But what if we want to dynamically change this number 42? That's where the CGI protocol (Common Gateway Interface) comes in again: we need to write a HTML CGI-Form where the end-user can enter the required number of centimetres and send that data to the web server so a dynamic query (or a parameterised query) can be performed which results in truly dynamic output.

Let's start on the server side first. For those of you who didn't read last month's column: start a new Web Module project by File | New, select Web Server Application from the repository and click OK to get the New Web Server Application Wizard. Here, you can select the required protocol to use: CGI, WinCGI or ISAPI/NSAPI. Like last time, I select either CGI or WinCGI, since I can test and debug these application more easily on my local machine with IntraBob version 2.01 (on last month's disk and also available on my website). After you select the protocol to use you get a new empty project with a Web Module.

Now, to get started for this time, simply drop a TQueryTableProducer from the Internet tab on the Web Module. This control will get connected to our dynamic query (or in fact our parameterised query, as we'll find out shortly), and generate the dynamic HTML pages from the query result. Right next to the TQueryTableProducer component, we need to put a TQuery component. Set the DatabaseName property to DBDEMOS and enter the following text in the SQL property:

```
SELECT * FROM BIOLIFE WHERE
 (BIOLIFE."Length (cm)" > :LENGTH)
```

This parameterised query will return all fields for every record in BIOLIFE where the field Length (cm) is greater than a run-time specified length, as passed in parameter LENGTH. Next, we need to specify the type of the LENGTH parameter by clicking on the Params property in the Object Inspector.

The type is Integer, and the default value is 0. We can test the SQL syntax by double-clicking on the Active property of the TQuery component. If it gets set to True, then we've built a valid SQL query.

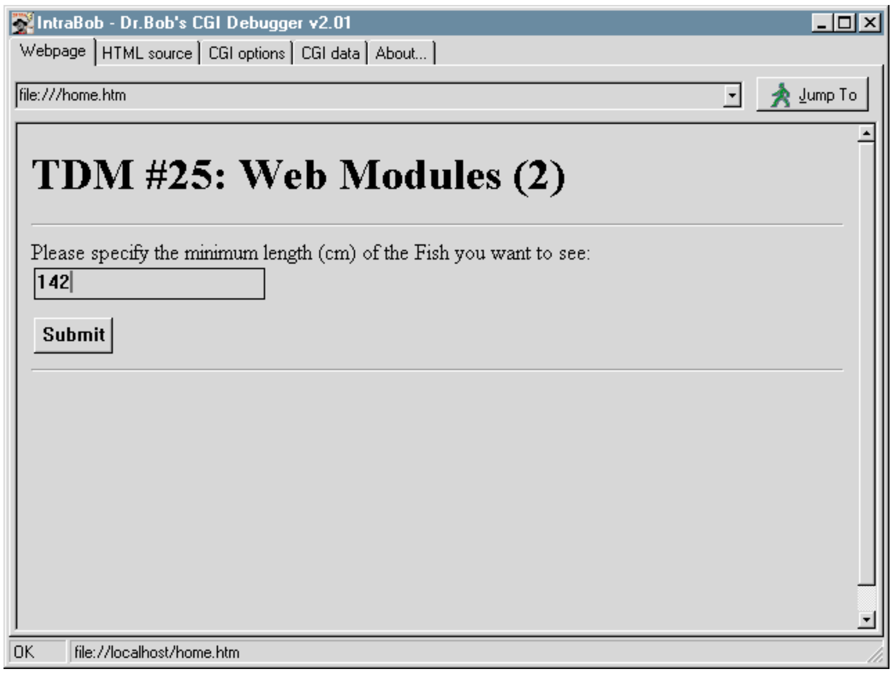
Now that we've entered the Query, it's time to connect it to the TQueryTableProducer. Just click on the TQueryTableProducer component on the Web Module, go to the Object Inspector and set the Query property to Query1. Now the QueryTableProducer will use the resulting database of the Query to generate the dynamic HTML pages. And that's not all. The QueryTableProducer is not only able to get the Query result, it is also able to set the Query input parameters (LENGTH in our case). For that, we need to write an HTML CGI Form that contains an INPUT field with name LENGTH (ie the name of the parameter inside the Query). Whenever the data of the Form is sent to the Web Server application, a match is made between the INPUT field names and the Query Parameter names. If a match is found, then the value of the INPUT field is automatically substituted for the Query parameter. In our case, this means we have to write a simple HTML CGI Form that contains an INPUT field with name LENGTH that should hold the length in centimetres of the records from the BIOLIFE table we want to see. This is where we might notice something funny about Delphi: we get all this great support for writing Web Modules with CGI/WinCGI and ISAPI/NSAPI support, but yet we have to write our own HTML CGI Forms to handle the input of these Web Modules. Of course, we could use IntraBuilder to generate the HTML forms for this purpose and connect them to the Delphi 3 Web Modules, but somehow that sounds somewhat less than obvious. For now, a HTML CGI Form that we can use is shown in Listing 1.

Note that the new Table-2-CGI Wizard is now available on my website (www.drbob42.com) which will automatically generate an HTML CGI Form similar to Listing 1, based on information from a table and feedback from the user (the names of the parameters and possible pre-filled values for fields).

All we need to do now is create a default `WebActionItem` (see last issue) and make sure the `OnAction` event contains the code in Listing 2 to redirect the output from the `TQueryTableProducer` to the final output of the web server app.

As long as we remember to keep the `Active` property of the `Query` set to `True`, there's no other code we need to write at this time. Let's just save it as `BIOLIFE`, compile the new project and try to run it with IntraBob v2.01. As long as we don't forget to specify the correct CGI-protocol (CGI or WinCGI), we only need to enter the minimum amount of centimetres, like 42, and hit the `SUBMIT` button. The result shows immediately (Figure 2).

Of course, we can use the `Columns` property (type `THTMLTableColumns`) of the `QueryTableProducer` just like we did with the `DataSetTableProducer` component in the last issue, and we end up in a new property editor in which we can design the output the way we want it.



TDM #25: Web Modules (2)

Please specify the minimum length (cm) of the Fish you want to see:

142

Submit

➤ *Figure 1*

```
<HTML>
<BODY>
<H1>TDM #25: Web Modules (2)</H1>
<HR>
<FORM ACTION="http://www.bolesian.nl/cgi_bin/biolife.exe" METHOD="POST">
<P>
Please specify the minimum length (cm) of the Fish you want to see:<BR>
<INPUT TYPE=text NAME="LENGTH">
<P>
<INPUT TYPE=SUBMIT>
</FORM>
<HR>
</BODY>
</HTML>
```

➤ *Listing 1*

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := QueryTableProducer1.Content;
end;
```

➤ *Listing 2*

State Of Independence

Now, say we don't want to view the entire output set all at once, but say we want to view only two records at a time, and be able to click on a `Next` button to view the next two records. For this, we need to be able to save some information, typically called "state information" (ie information that holds our current cursor and state of the query). There are at least two ways to save state information: using cookies, or using hidden fields in CGI Forms. Cookies are sent by the server to the browser, while CGI hidden fields are part of the HTML CGI Form and are sent by the client to the server as part of the next request. When using cookies, the initiative is with the web server, but the client has the ability to deny or disable a cookie. Servers sometimes even send cookies when you don't ask for them, which can be a reason why some people don't like cookies (like me, for example). Delphi 3 Web Modules do have built-in support for cookies, but I would like to explore the alternative with you: using hidden fields in HTML CGI Forms to save and communicate state information.

A hidden field is defined as any other data-entry field in a HTML CGI Form, except for the part that its type is "hidden", like:

```
<INPUT TYPE="hidden"
 NAME="KEY" VALUE="90020">
```

The hidden field above has the name `key` and value `90020` (which just happens to be a valid key value for the `BIOLIFE` table) and can be queried just like any other CGI Form input field (something we haven't done so far, but stay tuned, as we're about to).

We can save the `KEY` value of the current record and use it next time to start our query at that specific location (ie perform enough `Query.Next` operations until we're back at the specified `KEY` value again). This also means that we have to repeat our `Query` again (remember that a CGI or WinCGI application is executed and then terminated, so it's not just that HTTP is stateless, the application truly starts again, with no knowledge of any previous executions. And this of course yields the fact that we should also save the value of the `LENGTH` parameter to be able to correctly reconstruct the query for the subsequent executions.

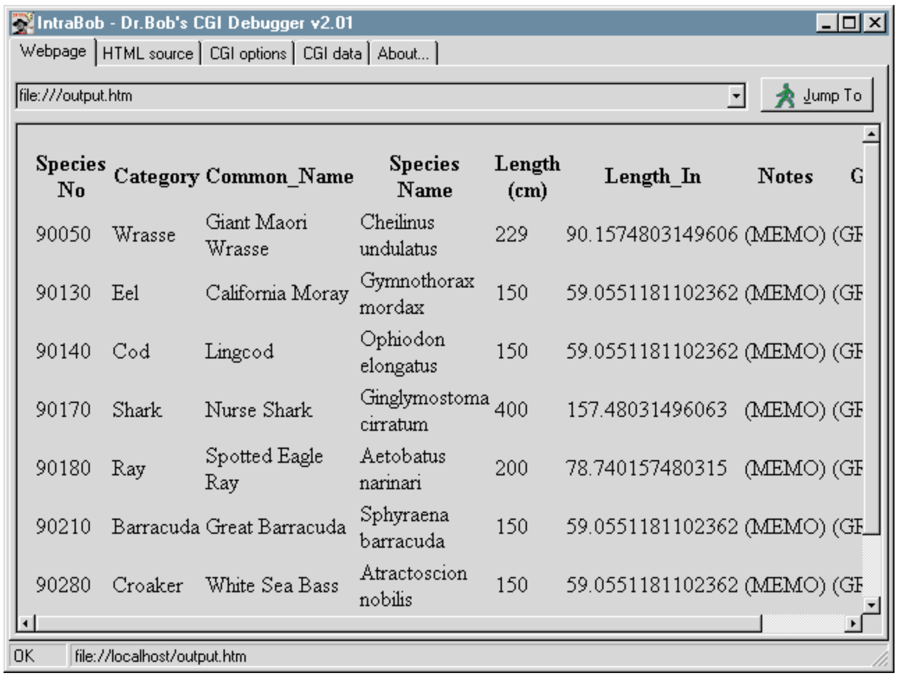
But first we need to limit the number of records in our generated HTML page to 2. For this, we need to set the `MaxRows` property of the `QueryTableProducer` from 20 to 2. Now, if we recompile the project and run it again, we only get the first two records of the `Query`. We need to do something more to be able to walk through the entire resultset of the `Query`.

Let's drop a `TPageProducer` on the Web Module, and edit the `HTMLDoc` property to contain the following lines:

```
<FORM ACTION="biolife.exe/query"
  METHOD="POST">
<INPUT TYPE=HIDDEN NAME="KEY"
  VALUE="<#KEY>">
<INPUT TYPE=HIDDEN NAME="LENGTH"
  VALUE="<#LENGTH>">
<P>
<INPUT TYPE=SUBMIT
  VALUE="Show next two records">
</FORM>
```

Note that this HTML document contains a CGI Form with the two required hidden input fields (one for the current `KEY` value and one for the `LENGTH` parameter for the query itself) and a Submit button (with caption `Show next two records`). The `VALUE` part of the hidden field conforms to the `TAG` syntax, so we can use the `OnTag` event of the `PageProducer` to replace the `KEY` tag with the value of the current key in the current record of the `Query`, as in Listing 3.

Note that we assume that `Query1` is `Active` (which it still is) and that `Fields[0]` holds the key value (which it in this case indeed does). In general, you may want to do a `FieldByName` or just insert the fields in the field editor and call them by name. Note, however, that this also works to place the value of the `Key` inside the dynamic `HTMLDoc` of the `PageProducer`.



IntraBob - Dr.Bob's CGI Debugger v2.01

Webpage | HTML source | CGI options | CGI data | About...

file:///output.htm — Jump To

Species No	Category	Common_Name	Species Name	Length (cm)	Length_In	Notes	G
90050	Wrasse	Giant Maori Wrasse	Cheilinus undulatus	229	90.1574803149606	(MEMO)	(GF
90130	Eel	California Moray	Gymnothorax mordax	150	59.0551181102362	(MEMO)	(GF
90140	Cod	Lingcod	Ophiodon elongatus	150	59.0551181102362	(MEMO)	(GF
90170	Shark	Nurse Shark	Ginglymostoma cirratum	400	157.48031496063	(MEMO)	(GF
90180	Ray	Spotted Eagle Ray	Aetobatus narinari	200	78.740157480315	(MEMO)	(GF
90210	Barracuda	Great Barracuda	Sphyraena barracuda	150	59.0551181102362	(MEMO)	(GF
90280	Croaker	White Sea Bass	Atractoscion nobilis	150	59.0551181102362	(MEMO)	(GF

OK | file://localhost/output.htm

➤ *Figure 2*

```
procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if TagString = 'KEY' then
    ReplaceText := Query1.Fields[0].AsString;
  if TagString = 'LENGTH' then
    ReplaceText := Query1.Params.ParamValues['LENGTH'];
end;
```

➤ *Listing 3*

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content :=
    QueryTableProducer1.Content +
    PageProducer1.Content;
end;
```

➤ *Listing 4*

Now, one of the nice features about the content of the different Delphi 3 C/S internet components is the fact that you can combine them into one. We can just add the `PageProducer1.Content` to the `QueryTableProducer1.Content` to get a bigger final `Response.Content` (see Listing 4).

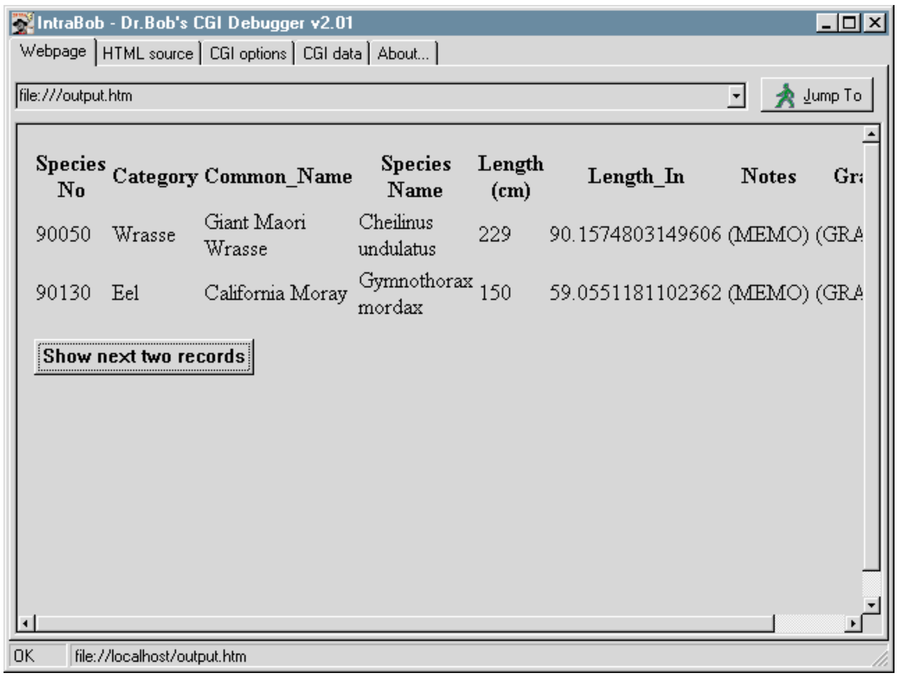
Note that since the `PageProducer1.Content` method is called after the `QueryTableProducer1.Content` could perform it's job, the `PageProducer` can actually get to the current `KEY` value of the `Query`, which is valid after the current `Query` records have been show (ie the next valid record).

Now, all we need to do is make sure that the above `Action` item can position the `Query` to the right position. For that, we need to be able to distinguish between a first time execution of a `Query` (with another value of the `LENGTH` parameter), and a subsequent request for record from the same `Query`. We can do this by using a different action, as was hinted in the `HTMLDoc` already (you did notice the `biolife.exe/query` `Action` part, didn't you?) combined with a `DataSetTableProducer` component with `MaxRows` properties set to the same value of 2.

So, we just go back to the Web Module, click on the `Actions` component editor, add a second action with `PathInfo` set to `/query` and enter the following code in the

```
procedure TWebModule1.WebModule1WebActionItem2Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var Key: Integer;
begin
  Query1.Close;
  Query1.Params[0].AsInteger := StrToInt(Request.ContentFields.Values['LENGTH']);
  try
    Key := StrToInt(Request.ContentFields.Values['KEY']);
  except
    Key := 0
  end { 0 for no key };
  Query1.Open;
  while Query1.Fields[0].AsInteger < Key do Query1.Next;
  { now create output page(s) }
  Response.Content := DataSetTableProducer1.Content + PageProducer1.Content;
end;
```

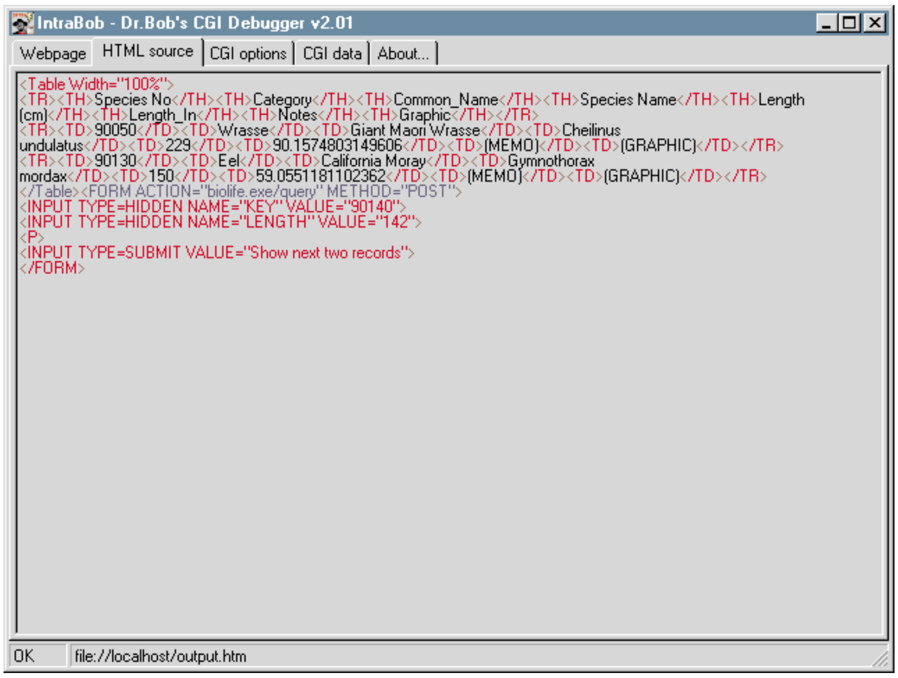
➤ *Listing 5*



Species No	Category	Common_Name	Species Name	Length (cm)	Length_In	Notes	Gra
90050	Wrasse	Giant Maori Wrasse	Cheilinus undulatus	229	90.1574803149606	(MEMO)	(GRA
90130	Eel	California Moray	Gymnothorax mordax	150	59.0551181102362	(MEMO)	(GRA

Show next two records

➤ *Figure 3*


IntraBob - Dr.Bob's CGI Debugger v2.01
Webpage
HTML source
CGI options
CGI data
About...
OK
file://localhost/output.htm

```
<Table Width="100%">
<TR><TH>Species No</TH><TH>Category</TH><TH>Common_Name</TH><TH>Species Name</TH><TH>Length
(cm)</TH><TH>Length_In</TH><TH>Notes</TH><TH>Graphic</TH></TR>
<TR><TD>90050</TD><TD>Wrasse</TD><TD>Giant Maori Wrasse</TD><TD>Cheilinus
undulatus</TD><TD>229</TD><TD>90.1574803149606</TD><TD>(MEMO)</TD><TD>(GRAPHIC)</TD></TR>
<TR><TD>90130</TD><TD>Eel</TD><TD>California Moray</TD><TD>Gymnothorax
mordax</TD><TD>150</TD><TD>59.0551181102362</TD><TD>(MEMO)</TD><TD>(GRAPHIC)</TD></TR>
</Table><FORM ACTION="biolife.exe/query" METHOD="POST">
<INPUT TYPE=HIDDEN NAME="KEY" VALUE="90140">
<INPUT TYPE=HIDDEN NAME="LENGTH" VALUE="142">
<P>
<INPUT TYPE=SUBMIT VALUE="Show next two records">
</FORM>
```

➤ *Figure 4*

OnAction event of this new Action. (See Listing 5).

First, we need to Close the Query and set the parameter LENGTH to the value that we can obtain from the TWebRequest parameter. TWebRequest has a ContentFields of type TStrings in case of a POST method and we can use the Values property to get the required values by name (such as Values('LENGTH') and Values('KEY')). Quite handy. After we've obtained the previous Key value and re-opened the Query, we only need to perform Next steps until we've reached (or exceeded) our previous Key value. After we've successfully positioned the Query cursor, it's time to produce the table content from the DataSetTableProducer1, followed by the PageProducer1 which again lists the Show next two records button and the two hidden fields, with the newly updated Key value, of course.

Now, why did we have to use a DataSetTableProducer1 in the first place? Couldn't we just re-use the same QueryTableProducer, if only to avoid having to set the Query LENGTH parameter by hand? No, we couldn't (I know, I tried), because the QueryTableProducer component will try to map every CGI data-entry field to a Query parameter. And while this works for the LENGTH parameter, the KEY field cannot be assigned to anything in the Query, so this actually results in a *Server Error 500*. Not funny. Maybe this might be a reason to try and derive from either the parent TDSTableProducer or the TQueryTableProducer itself, and try to make this parameter mapping mechanism a little more flexible.

If we take a look at the generated HTML source code (Figure 4), we can see the two hidden fields.

For each execution of the biolife.exe web application, a new KEY value is computed and entered in as hidden field in the HTML part that the PageProducer generates. And so is the LENGTH field repeated every time, so we're sure we're actually executing the same query over and over again (but just want to see different result set records).

Anyway, after all these exercises, the complete source code for the Web Module UNIT1.PAS is show in Listing 6.

A final reason why we have to use a TDataSetTableProducer and cannot use the TQueryTableProducer to show the next few records form a Query is the fact that the TQueryTableProducer assigns the query parameters (if any) and opens the query in the Contents method. In other words, while we may set the Query component to the exact position where we want it to be (like we can do for the TDataSetTableProducer), this won't work with the TQueryTableProducer which will just re-execute the query all over again.

Cookie Monster

Now that we know how to use hidden fields, can we do it a little bit differently as well? Yes, we can still use cookies if we want. In fact, the project source code on this month's disk will contain {$IFDEF} compiler directives to let you chose between using a hidden field or a cookie to send the current KEY value (the LENGTH query parameter field is always passed as hidden field, but I leave it up to you to pass this field as a cookie as well).

First of all, we need to set the initial value of the cookie in the WebActionItem1Action event. Cookies can be set as part of the Response, using the SetCookieField method. Like CGI values, a cookie is of the form NAME=VALUE, so we can put a KEY=value in there without much trouble (Listing 7).

Note that we're using a TStringList to set up a list of cookie values. Each list of cookies can have a Domain and Path associated with it, to indicate which URL the cookie should be sent to. You can leave these blank, of course. The fourth parameter specifies the expiration date of the cookie, which is set to Now+1 day, so next time the user is back the cookie will have expired. The final argument specifies whether or not the cookie is sent over a secure connection (which I just set to False). The above code results in the first cookie with value KEY=90140.

Watch closely, for the above cookie was actually the first cookie every programmatically *baked* by me. I'm still not convinced that cookies are the best solution in all cases, but it's good to know they're there when you need them...

Now, assuming the user accepts the cookie, then having set the cookie is still only half the work. In the WebModule1WebActionItem2Action event we need to read the value of the cookie, to determine how far to step with the Query to be able to show the next few records. In this case, cookies are part of the Request class, just like the ContentFields, and they can be queried using the CookieFields property (Listing 8).

Other than that, cookies work just like any (hidden or visual) content field. Just remember that while a content field is part of your

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, HTTPApp, Db, DBTables, DBWeb;
type
  TWebModule1 = class(TWebModule)
    QueryTableProducer1: TQueryTableProducer;
    Query1: TQuery;
    PageProducer1: TPageProducer;
    DataSetTableProducer1: TDataSetTableProducer;
    procedure WebModule1WebActionItem1Action(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
    procedure PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
      const TagString: String; TagParams: TStrings;
      var ReplaceText: String);
    procedure WebModule1WebActionItem2Action(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
  private
  public
  end;
var WebModule1: TWebModule1;
implementation
{$R *.DFM}
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := QueryTableProducer1.Content + PageProducer1.Content;
end;
procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if TagString = 'KEY' then
    ReplaceText := Query1.Fields[0].AsString;
  if TagString = 'LENGTH' then
    ReplaceText := Query1.Params.ParamValues['LENGTH'];
end;
procedure TWebModule1.WebModule1WebActionItem2Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var Key: Integer;
begin
  Query1.Close;
  Query1.Params[0].AsInteger := StrToInt(Request.ContentFields.Values['LENGTH']);
  try
    Key := StrToInt(Request.ContentFields.Values['KEY']);
  except
    Key := 0
  end { 0 for no key };
  Query1.Open;
  while Query1.Fields[0].AsInteger < Key do Query1.Next;
  { now create output page(s) }
  Response.Content :=
    DataSetTableProducer1.Content +
    PageProducer1.Content;
  end;
end.
```

➤ *Listing 6*

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
{$IFDEF COOKIES}
var
  Cookies: TStringList;
{$ENDIF}
begin
  Response.Content := QueryTableProducer1.Content + PageProducer1.Content;
{$IFDEF COOKIES}
  Cookies := TStringList.Create;
  Cookies.Add('KEY='+Query1.Fields[0].AsString);
  Response.SetCookieField(Cookies,'','',Now+1,False);
  Cookies.Free;
{$ENDIF}
end;
```

➤ *Listing 7*

request, so should always be up to date, a cookie may have been rejected, resulting in a possible older value (which was still on your disk a few sessions ago).

To test this, just run the BIOLIFE web application with cookies enabled, and (just for the fun of it) reject a cookie from time to time. You'll notice that if you reject the cookie, the old value will be used instead. So, I can only repeat that I prefer to use hidden fields, like we used in the first solution of this column.

ISAPI/NSAPI Notes

So far we've seen good results using CGI and WinCGI. These web applications, however, are just that: web *applications*. For every request they get started, execute and then terminate while returning their dynamic generated HTML pages. This usually takes more than one second, since loading the application, loading the BDE, performing the query, unloading everything just takes a while (even on a local machine). And while this may not sound too bad, imagine several people using the same CGI application at the same time. Don't think that won't happen.

My website gets almost 1,000 hits each day and I'm just doing this for fun (ie I am not a commercial website selling items), so it can get worse (or better, depending on your point of view). For my 1,000 hits each day, that means about 42 users each hour (give or take a few), or just under one user each

➤ *Listing 8*

```
procedure TWebModule1.WebModule1WebActionItem2Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  Key: Integer;
{$IFDEF COOKIES}
var
  Cookies: TStringList;
{$ENDIF}
begin
  Query1.Close;
  Query1.Params[0].AsInteger := StrToInt(Request.ContentFields.Values['LENGTH']);
  try
  {$IFDEF COOKIES}
    Key := StrToInt(Request.CookieFields.Values['KEY']);
  {$ELSE}
    Key := StrToInt(Request.ContentFields.Values['KEY']);
  {$ENDIF}
  except
    Key := 0
  end { 0 for no key };
  Query1.Open;
  while Query1.Fields[0].AsInteger < Key do Query1.Next;
  { now create output page(s) }
  Response.Content := DataSetTableProducer1.Content + PageProducer1.Content;
{$IFDEF COOKIES}
  Cookies := TStringList.Create;
  Cookies.Add('KEY='+Query1.Fields[0].AsString);
  Response.SetCookieField(Cookies,'','',Now+1,False);
  Cookies.Free;
{$ENDIF}
end;
```

minute. Considering the fact that most users will spend more than a few minutes at my site (maybe even ten or fifteen minutes), I can predict that at any given time, at least a few users, like a dozen, will be simultaneously browsing my website. While not all of them will be using my CGI applications, every one of them will at least fire my new Hit Counter (also a CGI application), so my website and its users will not go unnoticed by my WinNT web server.

Of course, we can use ISAPI or NSAPI web server extension DLLs, which are not just stand-alone applications but DLLs that get loaded only once and unload when the server gets down. These avoid the time you need to load and unload your web application, and can also avoid (re-)initialisation of the BDE. This last issue needs a little more attention: when using the BDE in an ISAPI/NSAPI DLL, you're in fact offering the web surfers a multi-threaded web server application they can use: while each CGI application gets loaded for every request, there is only one instance of the ISAPI/NSAPI DLL loaded. To avoid problems with session names, one should give each thread its own session, by adding a `TSession` component to the Web Module and setting the `AutoSession` property to `True`. This will ensure that each thread will get a session with a unique name generated.

Next Time...

Next month, we'll use the information from this month's column to write a final Web Module sample application, the one I mentioned a little bit earlier: a homepage "hit" counter, complete with registration and statistical analysis (check my website at www.drbob42.com for a preview of the results). Both a Web Module and a non Web Module Delphi 3 version will be available, showing the differences in ease-of-use, maintenance, code size, speed, etc.

Also next time Chad Hower will have the promised article about Portcullis, the Internet Application Gateway from ShoreLine, which will specifically be about how to write IAG-compatible components (heavy work commitments meant I had to bow out, so Chad's on his own for this one!).

Bob Swart (aka Dr.Bob, visit www.drbob42.com) is a professional knowledge engineer technical consultant using Delphi and Borland C++Builder for Bolesian (www.bolesian.com), a freelance technical author for *The Delphi Magazine* 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 and intranet. In his spare time, Bob likes to watch video tapes of *Star Trek Voyager* and *Deep Space Nine* with his 3-year old son Erik Mark Pascal and his 9-month old daughter Natasha Louise Delphine.