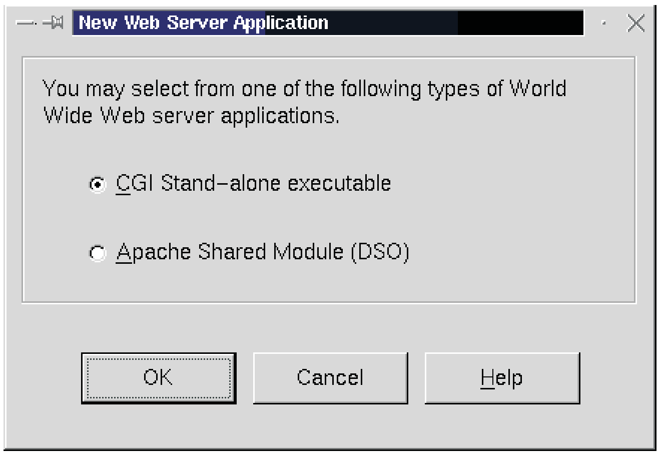
Under Construction: WebBroker For Linux

by Bob Swart

In this article, we'll see how we can use Kylix to create web server applications for the Apache web server running on Linux.

First of all, I'll assume that you have Kylix installed, as well as the Apache web server. You will, I'm afraid, need the Server Developer edition of Kylix. Take heart, though, as you can also develop web applications using the Desktop Developer edition of Kylix, as console applications, in the same way that you have been able to in Delphi. I'll show you how to do this next month.

Start Kylix and close the default project. Start a new project using `File | New`, and select the `Web Server Application` icon from the Object Repository. This will show you the `New Web Server Application` wizard (Figure 1).


New Web Server Application

You may select from one of the following types of World Wide Web server applications.

- [✓ CGI Stand-alone executable]
- Apache Shared Module (DSO)

➤ *Figure 1*

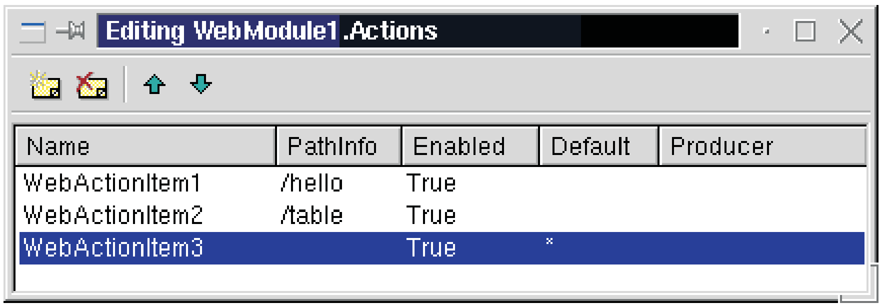
Select the default choice here for a CGI standalone executable. This will generate a new Web Broker project. We need to save this project, so select `File | Save All`, which will prompt you for the filename of the web module (which I call webmod.pas) and the filename of the main project (which I call cgi42.dpr in this case).

Now click on the empty web module called `WebModule1`. This is the central point of our web server application. It looks very much like a data module, and is in fact a data module with a special part added to it: a `TWebDispatcher` component.

This means that if you don't want to start with an empty web module, but want to continue working with an existing data module, you can do so by dropping a `WebDispatcher` component from the `Internet` tab of the component palette onto the data module to turn it into a web module. The Web Dispatcher will analyse incoming client requests, and dispatch each request to a corresponding `WebActionItem`. The consequence of this design (a web dispatcher and a collection of `WebActionItems`) is that a single web server application can actually perform multiple tasks. We can have a `WebActionItem` that can be triggered to generate an overview of information (for example, items from a catalogue), another one to order a specific item, and a last one to present the invoice. All three can be combined in a single application, thereby sharing everything they have in common (such as database connectivity, layout specifics, etc).

For the example in this article, we'll create three `WebActionItems`. First, right-click on the web module to start the Action Editor. Then, click three times on the yellow icon to create three new `WebActionItems`. For each one, we can use the Object Inspector to configure them. This is needed because `WebActionItems` are distinguished by their `PathInfo` property (the information appended to the URL, as we'll see in a moment). In this case, let's give the first `WebActionItem` a `PathInfo` value of `/hello`, the second one `/table` and leave the third one empty. We also need to specify a default `WebActionItem` (the one which will be selected by the Web Dispatcher if none of the `PathInfo` values match), and I always take the last one, with the empty `PathInfo` as default `WebActionItem` (note that once it's the default, the `PathInfo` no longer matters), see Figure 2.

➤ *Figure 2*

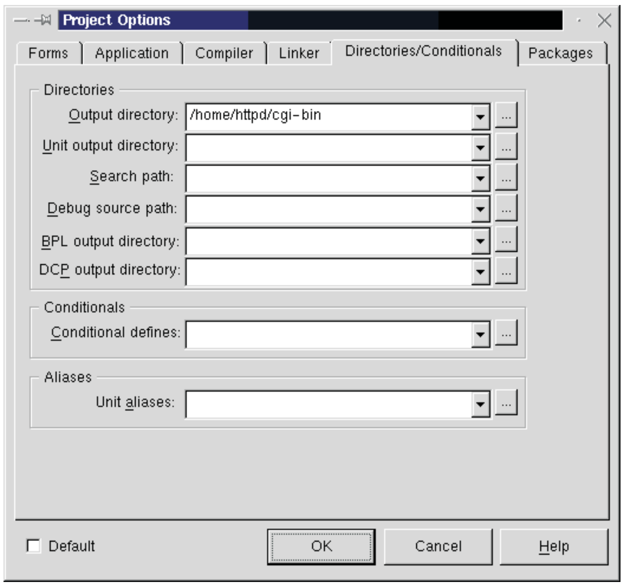

Editing WebModule1.Actions

Name	PathInfo	Enabled	Default	Producer
WebActionItem1	/hello	True		
WebActionItem2	/table	True		
WebActionItem3		True	*	

Select the default `WebActionItem` (with `/hello PathInfo`), and switch to the Object Inspector again. Click on the `Events` tab, and double-click on the `OnAction` event to generate the event handler code in the editor. Inside the editor, you can now write the response code for this default `WebActionItem`. Apart from the first `Sender` parameter, the `OnAction` event handler has three more parameters: `Request`, `Response` and `Handled`. If we ignore the last two for now, we can use the `Response` to return something for this default action. In this case, I just want to say *Made in Kylix!*, which I can return in the `Response.Content` property. In short, my event handler is coded as in Listing 1.

We're almost ready with the first web server application written in Kylix. It's now time to make sure the resulting application is positioned in the right directory, so Apache can find it and execute it. On my machine, working as root, I have a /home/httpd/cgi-bin directory that can contain CGI applications for the Apache web server. To make sure that cgi42 ends up in this cgi-bin directory, we need to specify the `Output Path` in the `Directories/Conditionals` tab of the Project Options dialog (see Figure 3).

Figure 3



Directories
Output directory: /home/httpd/cgi-bin
Unit output directory:
Search path:
Debug source path:
BPL output directory:
DCP output directory:
Conditionals
Conditional defines:
Aliases
Unit aliases:
Default

Apache

We can now compile the cgi42 application, and it will indeed result in a cgi42 executable file inside the /home/httpd/cgi-bin directory. However, before we can execute it, we first need to tell Apache where to find the libraries that Kylix Web Broker applications need. For this, we need to manually edit the httpd.conf file as follows:

```
vi /etc/httpd/conf/httpd.conf
```

Add a single line to the end of the file, with the following content:

```
SetEnv LD_LIBRARY_PATH
  /root/kylix/bin
```

If you are deploying your application on another machine (where Kylix is not installed), see the *Kylix WebBroker Deployment* boxout later. After you have modified the httpd.conf file, you will need to explicitly restart the Apache web server using the following command:

```
/etc/rc.d/init.d/ httpd restart
```

At this time, we're ready to start our browser and show the cgi42 web server application. The easiest way is to call http://localhost/cgi-bin/cgi42, or connect to my Linux machine from another machine using the IP address, for example http://192.168.92.244/cgi-bin/cgi42 and watch the results (Figure 4).

This concludes the first native Linux web server application written in Kylix. The nice thing to notice is that, apart from configuring Apache to find the supporting libraries, we didn't do anything that we wouldn't have do to when using Delphi. In fact, I could take the source code from this project and recompile it on my Windows machine to produce a regular Windows CGI application.

PageProducers

Let's return to Kylix, and open up the Web Module again. This time, we're going to use some help to produce HTML code. Specifically, there are `PageProducer` and `Table-Producer` components on the Internet tab of the component palette that are very powerful to use. Let's start with a `PageProducer` component first, and drop one from the Internet tab on the web module.

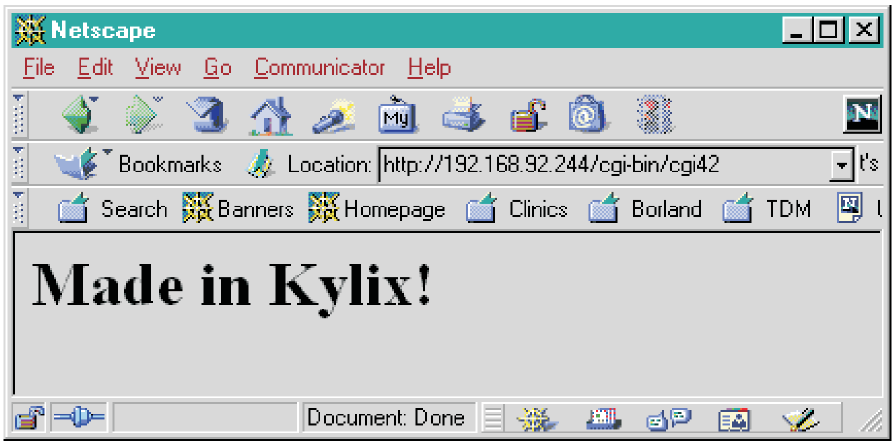
A `PageProducer` component can be used to prepare an HTML template file (inside either the built-in `HTMLDoc` property or the external pointing `HTMLFile` property). For this example, I'm using the `HTMLDoc` property, but in real life the `HTMLFile` property may be more powerful, since it enables you to maintain the HTML template without having to recompile the web server application itself, which can be quite handy if you have separate web design and web application teams.

Anyway, inside the String List Editor for the `HTMLDoc` property, I can specify the HTML template, including some special non-HTML

Listing 1

```
procedure TWebModule1.WebModule1Actions2Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '<H1>Made in Kylix!</H1>';
end;
```

Figure 4



http://192.168.92.244/cgi-bin/cgi42

Made in Kylix!

Document: Done

tags using the # character (I usually pronounce this as the 'hash' character, and not the 'pound' character, but recent events have led me to believe I should probably call it the 'sharp' character instead). These so-called #-tags are special placeholders that each trigger an event handler to dynamically replace the #-tag with another value.

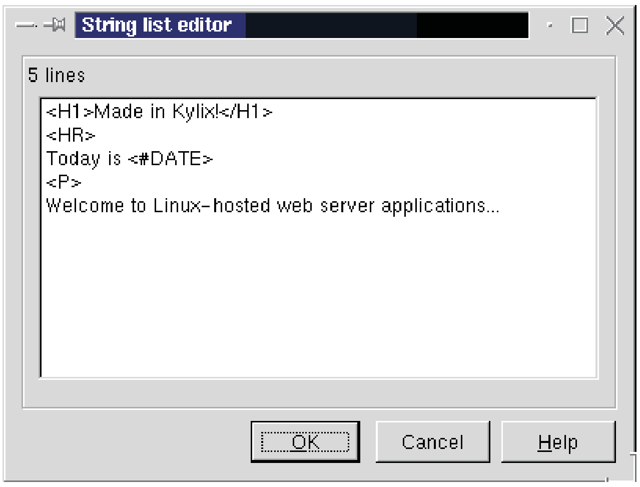
In this case (Figure 5), I clearly wish to replace the <#DATE> tag with the current date (and time, to show that it is indeed updated dynamically once you refresh the page).

Close the String List Editor and move to the Events tab of the Object Inspector to see the OnHTMLTag event that is defined for the PageProducer component. This event handler will be fired for every #-tag inside the HTMLDoc (or inside the external file specified by the value of HTMLFile, see Listing 2).

Now, hit F12 to display the Web Module again. Right-click inside the Web Module to start the Action Editor. Select the first WebActionItem (the one with the /hello PathInfo), and go to the Object Inspector. Here, click on the Properties tab. Unlike the previous time we handled a WebActionItem, we do not write code for the OnAction event handler, but use the Producer property instead. This property can point to any page producer or table producer, which can help in generating HTML for us. In this case, there is only one PageProducer, of course, so we just select that one. Now, recompile the application, and enter the following URL inside your browser:

```
http://localhost/cgi-bin/
  cgi42/hello
```

As you can see, it's the same URL as before, but with the /hello PathInfo appended to it. The result is shown in Figure 6. As you can see, the PathInfo /hello is appended to the URL as we have used before. This is a convenient way to request different actions from the same application.



```
<H1>Made in Kylix!</H1>
<HR>
Today is <#DATE>
<P>
Welcome to Linux-hosted web server applications...
```

➤ *Figure 5*

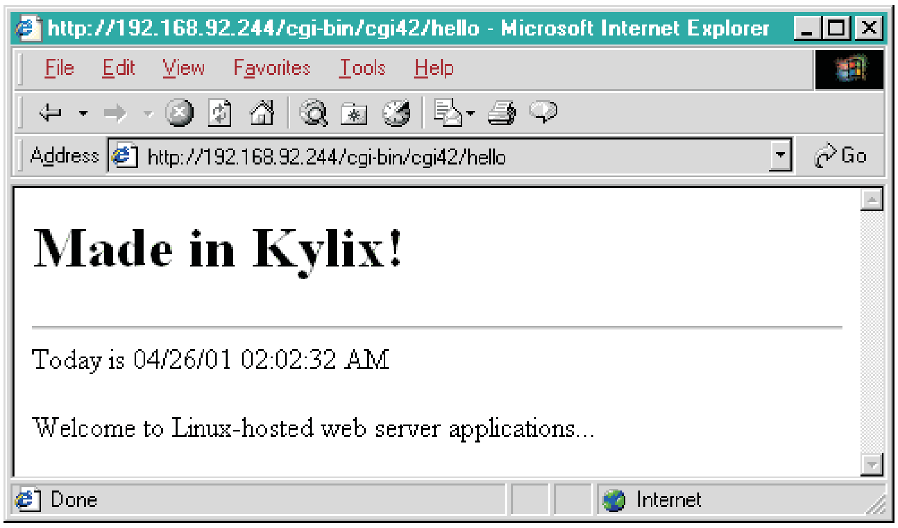
```
procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if TagString = 'DATE' then
    ReplaceText := DateTimeToStr(Now);
end;
```

➤ *Above: Listing 2*

➤ *Below: Listing 3*

```
procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if TagString = 'DATE' then
    ReplaceText := DateTimeToStr(Now)
  else
    if TagString = 'counter' then
      ReplaceText := 'user #' + Request.CookieFields.Values['counter'];
end;
```

➤ *Figure 6*


http://192.168.92.244/cgi-bin/cgi42/hello - Microsoft Internet Explorer
File
Edit
View
Favorites
Tools
Help
Address
http://192.168.92.244/cgi-bin/cgi42/hello
Go
Done
Internet

Made in Kylix!

Today is 04/26/01 02:02:32 AM

Welcome to Linux-hosted web server applications...

Cookies

WebBroker applications have full support for cookie functionality to maintain state among sessions, or implement local user configuration information. A simple example of using a cookie is a counter, which is stored in the local cookie file, and can be used to display the number of visits by a single user to a website.

In order to display a cookie value, we need to change the HTMLDoc property of the PageProducer, and add a special <#counter> tag (Figure 7). Next, we need to extend the OnHTMLTag event handler as in Listing 3. We also need to set the cookie value (and increase it after each visit), which can best be done in the OnAction event handler for the /hello ActionItem, as in Listing 4. Note that we now use the OnAction event handler as well as the Producer property of this WebActionItem. That's perfectly normal, as both get 'executed', starting with the Producer property first. The result is a counting cookie (Figure 8).

User Input

Apart from CookieFields, we can also work with user input. Two collections are offered to help us: ContentFields (for the POST protocol) and QueryFields (for the GET protocol). Unfortunately, the first release of Kylix Server Edition contains a bug which causes the ContentFields to be left empty.

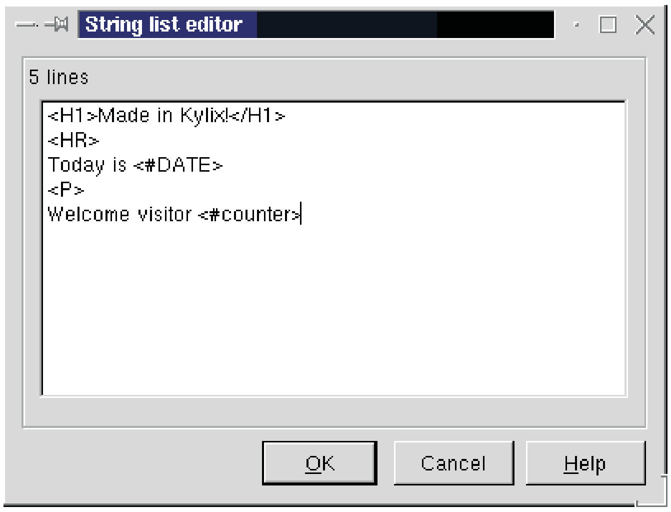
In order to verify this, let's make a form using a simple input field called login, as shown in Listing 5. Save this form in a file called login.htm so we can use it later. The content of the HTMLDoc property is changed to include a new #-tag for the login name, as follows:

```
<H1>Made in Kylix!</H1>
<HR>
Today is <#DATE>
<P>
Welcome visitor <#counter>
  (<#login>)
```

Now we need to extend the OnHTMLTag event handler again, as shown in Listing 6. Note the QueryFields, which belong to the GET protocol. Using the GET protocol means that all the input fields are passed on the URL, which means that if we enter Bob in the edit field and then click on the Submit button, the URL will actually contain the login=Bob string:

```
http://192.168.92.244/cgi-bin/
  cgi42/hello?login=Bob
```

When using METHOD=POST in the form, we must replace QueryFields with ContentFields. However, this exposes a bug in the current release of Kylix, since it appears that the ContentFields are not correctly initialised. This problem will be fixed in an upcoming update pack for Kylix (so stay tuned to my website at www.drbob42.com/kylix to find out when this will appear). In the meantime, we can use the GET protocol and switch over to using POST later, since the WebAction items themselves respond to either protocol. You only need to modify QueryFields (for GET) into ContentFields (for POST) when this bug is fixed. Or stick with GET, but I always prefer POST as it looks (and feels) much 'cleaner' with no ugly fat URLs.



```
<H1>Made in Kylix!</H1>
<HR>
Today is <#DATE>
<P>
Welcome visitor <#counter>
```

➤ Figure 7

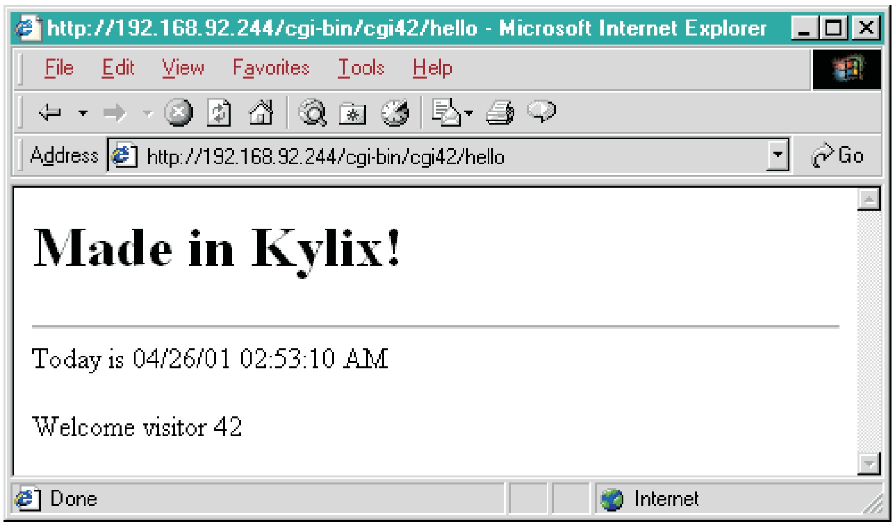
```
procedure TWebModule1.WebModule1Actions0Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  Cookie: Integer;
  Cookies: TStringList;
begin
  Cookie := StrToIntDef(Request.CookieFields.Values['counter'],0);
  Inc(Cookie);
  Cookies := TStringList.Create;
  try
    Cookies.Add('counter=' + IntToStr(Cookie));
    Response.SetCookieField(Cookies,'','',-1,false);
  finally
    Cookies.Free
  end
end;
```

➤ Above: Listing 4

➤ Below: Listing 5

```
<HTML>
<BODY>
<FORM ACTION="http://192.168.92.244/cgi-bin/cgi42/hello" METHOD=GET>
  Login: <INPUT TYPE=text NAME=login>
<BR>
<INPUT TYPE=submit>
</FORM>
</BODY>
</HTML>
```

➤ Figure 8


http://192.168.92.244/cgi-bin/cgi42/hello - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://192.168.92.244/cgi-bin/cgi42/hello
Go
Done
Internet

Made in Kylix!

Today is 04/26/01 02:53:10 AM

Welcome visitor 42

Tables In HTML

And now for the final demo: showing a database table using a unidirectional dataset. First, drop a TSQLConnection component on the web module. Set IBLocal as ConnectionName, and make sure to set the LoginPrompt property to False (otherwise your web server would get the login dialog, and the user would get nothing). Next, drop a `TSQLQuery` component, set its `SQLConnection` property to the `SQLConnection1` component, and write the following line of SQL for the `SQL` property:

```
select * from employee
```

Activate the query to make sure everything is fine.

Now, go back to the `Internet` tab of the Component Palette, and drop a `TDataSetTableProducer` component on the web module. This component can work with any dataset, including `TSQLTable`, `TSQLQuery` and `TSQLStoredProc` (as well as the `TSQLClientDataSet`), but in this case a unidirectional dataset is more than enough). Assign the `DataSet` property of the `DataSetTableProducer` component to the `SQLQuery` component, and prepare yourself to do some design settings.

Using Delphi in Windows, we could either right-click on the `DataSetTableProducer` and select

Figure 9

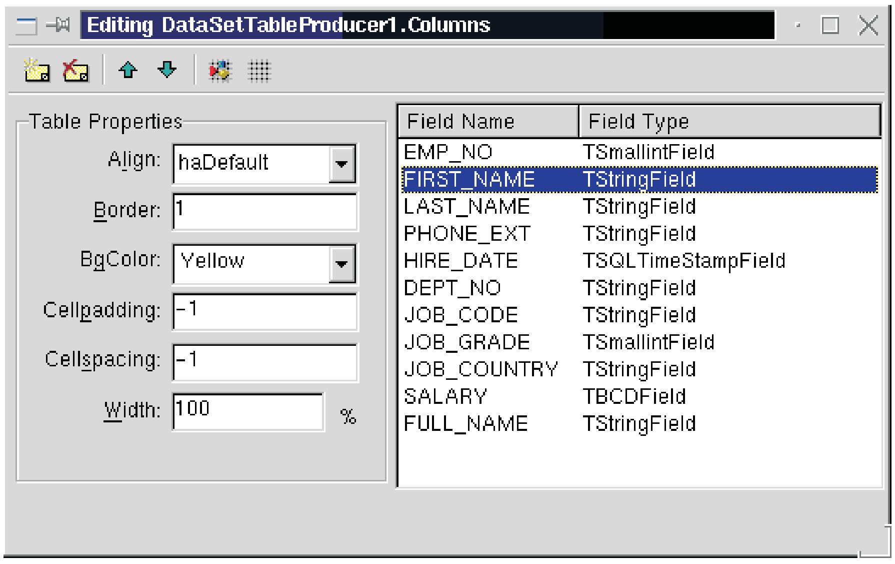

Editing DataSetTableProducer1.Columns

Table Properties

Align:	haDefault
Border:	1
BgColor:	Yellow
Cellpadding:	-1
Cellspacing:	-1
Width:	100 %

Field Name	Field Type
EMP_NO	TSmallintField
FIRST_NAME	TStringField
LAST_NAME	TStringField
PHONE_EXT	TStringField
HIRE_DATE	TSQLTimeStampField
DEPT_NO	TStringField
JOB_CODE	TStringField
JOB_GRADE	TSmallintField
JOB_COUNTRY	TStringField
SALARY	TBCDField
FULL_NAME	TStringField

Listing 6

```
procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TString; var ReplaceText: String);
begin
  if TagString = 'DATE' then
    ReplaceText := DateTimeToStr(Now)
  else
    if TagString = 'counter' then
      ReplaceText := 'user #' + Request.CookieFields.Values['counter']
    else
      ReplaceText := Request.QueryFields.Values['login']
end;
```

Listing 7

```
unit webmod;
interface
uses
  Variants, SysUtils, Classes, HTTPApp, HTTPProd,
  DBXpress, FMTBcd, DBWeb, DB, SqlExpr;
type
  TWebModule1 = class(TWebModule)
    PageProducer1: TPageProducer;
    SQLConnection1: TSQLConnection;
    SQLQuery1: TSQLQuery;
    DataSetTableProducer1: TDataSetTableProducer;
    procedure WebModule1Actions2Action(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var
      Handled: Boolean);
    procedure PageProducer1HTMLTag(Sender: TObject; Tag:
      TTag; const TagString: String; TagParams: TStrings;
      var ReplaceText: String);
    procedure WebModule1Actions0Action(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var
      Handled: Boolean);
    procedure WebModule1Actions1Action(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var
      Handled: Boolean);
  private
  public
  end;
var
  WebModule1: TWebModule1;
implementation
uses
  WebReq;
{$R *.xfm}
procedure TWebModule1.WebModule1Actions2Action(Sender:
  TObject; Request: TWebRequest; Response: TWebResponse;
  var Handled: Boolean);
begin
  Response.Content := '<H1>Made in Kylix!</H1>';
end;
procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject;
  Tag: TTag; const TagString: String; TagParams: TStrings;
  var ReplaceText: String);
begin
  if TagString = 'DATE' then
    ReplaceText := DateTimeToStr(Now)
  else if TagString = 'counter' then
    ReplaceText := Request.CookieFields.Values['counter']
  else // login
    ReplaceText := Request.QueryFields.Values['login']
    // Request.QueryFields == GET protocol (fat URL)
    // Request.ContentFields for POST doesn't work!
end;
procedure TWebModule1.WebModule1Actions0Action(Sender:
  TObject; Request: TWebRequest; Response: TWebResponse;
  var Handled: Boolean);
var
  Cookie: Integer;
  Cookies: TStringList;
begin
  Cookie :=
    StrToIntDef(Request.CookieFields.Values['counter'],0);
  Inc(Cookie);
  Cookies := TStringList.Create;
  try
    Cookies.Add('counter=' + IntToStr(Cookie));
    Response.SetCookieField(Cookies,'','',-1,false);
  finally
    Cookies.Free
  end
end;
procedure TWebModule1.WebModule1Actions1Action(Sender:
  TObject; Request: TWebRequest; Response: TWebResponse;
  var Handled: Boolean);
begin
  try
    // SQLConnection1.LoadParamsOnConnect := False;
    SQLConnection1.Connected := True;
    try
      Response.Content :=
        DataSetTableProducer1.Content
    finally
      SQLConnection1.Connected := True
    end
  except
    on E: Exception do
      Response.Content := 'Error: ' +
        E.Message
  end
end;
initialization
  WebRequestHandler.WebModuleClass := TWebModule1;
end.
```

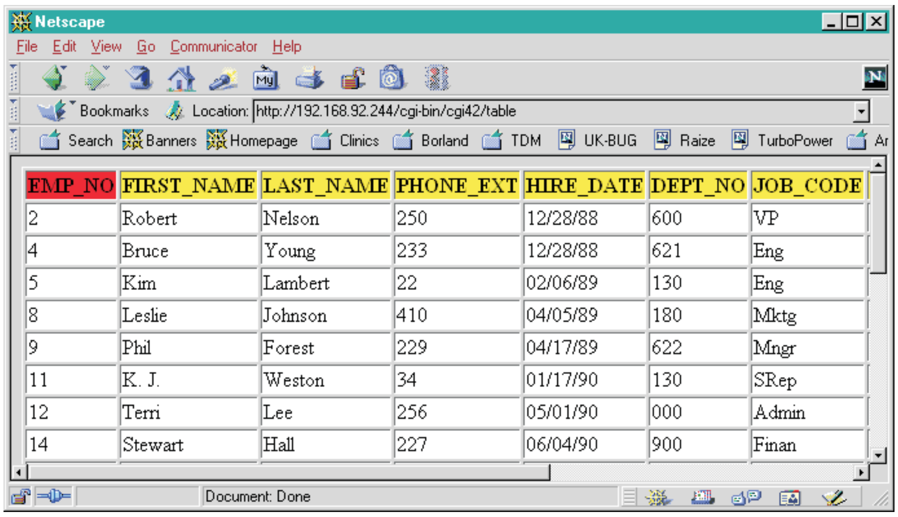
the Response Editor option, or click on the ellipsis (...) for the `Columns` property (of the `DataSet-TableProducer`) to get the Response Editor dialog containing a WYSIWYG preview. This preview was based on the Internet Explorer ActiveX control, and I was wondering what the support under Linux would look like. It turns out we still get a dialog to edit the properties of the different table columns (ie all the fields in the dataset), but no longer see a preview of the result. Maybe next release of Kylix?

Regardless of the preview ability, the Columns Editor Dialog is still very powerful, as it allows us to set the global table options such as cell alignment and padding, border, background colour and table width. Note that these are options for the entire table! (Figure 9).

For individual options (for each of the table columns) we still need to select an individual column and move to the Object Inspector. Here, we can still set individual column options such as the background colour, alignment (both horizontal and vertical), and even individual options for the column header.

After we're done customising the HTML options, we should make sure the `DataSetTableProducer` is indeed used by connecting it to the `/table` action item. I've written the code in the `OnAction` event handler (see Listing 7).

➤ *Figure 10*


Netscape
File Edit View Go Communicator Help
Bookmarks
Location: http://192.168.92.244/cgi-bin/cgi42/table
Search Banners Homepage Clinics Borland TDM UK-BUG Raize TurboPower
Document: Done

EMP_NO	FIRST_NAME	LAST_NAME	PHONE_EXT	HIRE_DATE	DEPT_NO	JOB_CODE
2	Robert	Nelson	250	12/28/88	600	VP
4	Bruce	Young	233	12/28/88	621	Eng
5	Kim	Lambert	22	02/06/89	130	Eng
8	Leslie	Johnson	410	04/05/89	180	Mktg
9	Phil	Forest	229	04/17/89	622	Mngr
11	K. J.	Weston	34	01/17/90	130	SRep
12	Terri	Lee	256	05/01/90	000	Admin
14	Stewart	Hall	227	06/04/90	900	Finan

Kylix WebBroker Deployment

When deploying Kylix web server applications (on Apache web servers for example), you generally only need to distribute your CGI executable or shared object (DSO) file.

However, remember that we had to set the LD_LIBRARY_PATH to /root/kylix/bin in the httpd.conf file? This means that potentially other files must be deployed as well, such as specific Kylix runtime packages. When using dbExpress, for example, you need to deploy the specific driver for the database that you're using (like libsqlib.so for InterBase, libsqlmys.so for MySQL, libsqlora.so for Oracle, or libsqldb2.so for DB2). Also, when using ClientDataSets or SQLClientDataSets, you must not forget to deploy midas.so as well.

See Kylix Developer's Guide page 13-5 and the file "deploy" in your Kylix directory for more information.

But before we can test the new web broker application, we must first make sure that everything is set right to access dbExpress from our CGI application. As Brian Long described in his *Apache Shared Modules* article last month, we need to add two more lines to the /etc/httpd/conf/httpd.conf file:

```
SetEnv LANG en_US
SetEnv HOME /home/bswart
```

Note that if you installed Kylix as root, you need to set the `HOME` to /root instead (or any other place where a .borland hidden directory can be found).

After we have made those last modifications, we are ready to make a last compile and test the web server application using dbExpress. The result can be shown in any browser again, and looks like Figure 10.

The final source code for all the examples in this column can be seen in Listing 7.

Note that the code from the final listing will no longer compile using Delphi 5 (because we used dbExpress, which is not available in Delphi 5), but it will probably work without a single source code change with the new Delphi 6.

Next Time

So far we've seen web server application support in Kylix Server Edition using WebBroker technology inside NetCLX. However, as I said, you need Kylix Server Edition for these tricks, which is the current high-end (and most expensive) edition of Kylix. Can't we produce web server applications with the Kylix Desktop Edition? Or with the free and forthcoming Open Edition? Sure we can, just not with WebBroker, but the 'hard way'. Which can also be fun, and very enlightening. So next time, join me as I show you how to write web server applications with Kylix Desktop edition, without WebBroker, *with* dbExpress, and still with great results. *So stay tuned...*

Bob Swart (aka Dr.Bob, visit www.drbob42.com) is an @- Consultant, Trainer and co-founder of the Kylix/Delphi Oplossings-Centrum (see www.KDOC.nl), a PinkRoccade nv Company in The Netherlands.