Under Construction:
Kylix Desktop Does CGI Too

by Bob Swart

In this article, we'll see how we can use Kylix Desktop Developer to create web server applications for the Apache web server running on Linux. Remember that Kylix Server Developer is the (expensive) high-end version, which is officially sold as the version to use when creating web server applications. But Kylix Desktop Developer can also do this, and its price was recently slashed to £139 in the UK and $199 in the USA *[This is supposed to be a limited time offer, but I for one think Borland would be plain crazy to raise the price again. Ed]*.

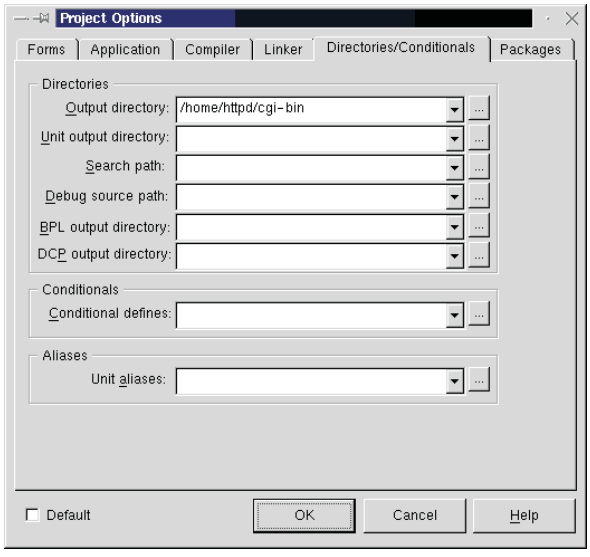
Console Applications

First of all, I shall assume that you have Kylix installed on your machine, as well as the Apache web server (like last time when we used WebBroker in Kylix Server Developer). Start Kylix, and close the default project. For web server applications 'the hard way' we need a console application, so choose `File | New` and select the `Console ...` icon (on my machine, I don't see the full title). This will give you a new empty console application, consisting of just four lines of code:

```
program Project1;
{$APPTYPE CONSOLE}
begin
end.
```

While this is certainly not earth-shattering, when you compile this application, you will see that the minimum do-nothing application in Kylix generates 14,916 bytes of machine code (14,908 bytes if you remove the `{$APPTYPE CONSOLE}` line too). This program does nothing, of course, but it serves as a reminder to make the point that web applications developed this way are really small compared to WebBroker applications. And small means fast and resource-efficient.

➤ *Figure 1: Kylix Project Options (Kylix on Linux).*



CGI Applications

Web server applications come in different types. The easiest type to make is CGI, which stands for Common Gateway Interface: the communication protocol between a form on a web browser page (the client) and an application running on the web server (the server). The application is usually called a CGI script (when written in an interpreted language like Perl) or CGI application (when written in a compiled language like C/C++, or a development environment like Delphi or Kylix).

```
program hello;
{$APPTYPE CONSOLE}
begin
  writeln('content-type: text/html');
  writeln;
  writeln('<html>');
  writeln('<body bgcolor=ffffcc>');
  writeln('<h1>Made in Kylix Desktop!</h1>');
  writeln('</body>');
  writeln('</html>')
end.
```

➤ *Listing 1: CGI application hello.dpr.*

A CGI application interacts with the user (who inputs data in a browser) by reading the standard input, and writing to standard output. The output of a CGI application usually consists of HTML, but we can generate other results as well, such as images or even streaming audio or video.

Content And Type

Before we can generate HTML (or any other output format, such as binary image data), we first have to return the MIME content-type, followed by an empty line. In the case of HTML, that can be done as follows:

```
writeln(
  'content-type: text/html');
writeln;
```

Obviously, other content-types are possible, such as `image/gif` or `image/jpeg`, always followed by an empty line before the data itself (binary in the case of images: a topic for a later article on web server development with Kylix).

After we have set the content-type, it's time to return the actual content. Note that this includes the `<html>` and `<body>` opening and closing tags. The simplest example is to show the

line *Made in Kylix Desktop Developer!* Just like the WebBroker application which we made in Kylix Server Developer last month. The source code is in Listing 1.

It won't be a surprise (I hope) when I tell you that the simple CGI application from Listing 1 compiles with Kylix as well as any version of Delphi (so you can use it to produce a CGI application for a Windows web server as well as a Linux web server). In fact, a lot of what I'm doing in this article will work in Delphi on Windows as well as Kylix on Linux (although the focus will be on support for Kylix Desktop on Linux, as you'll see).

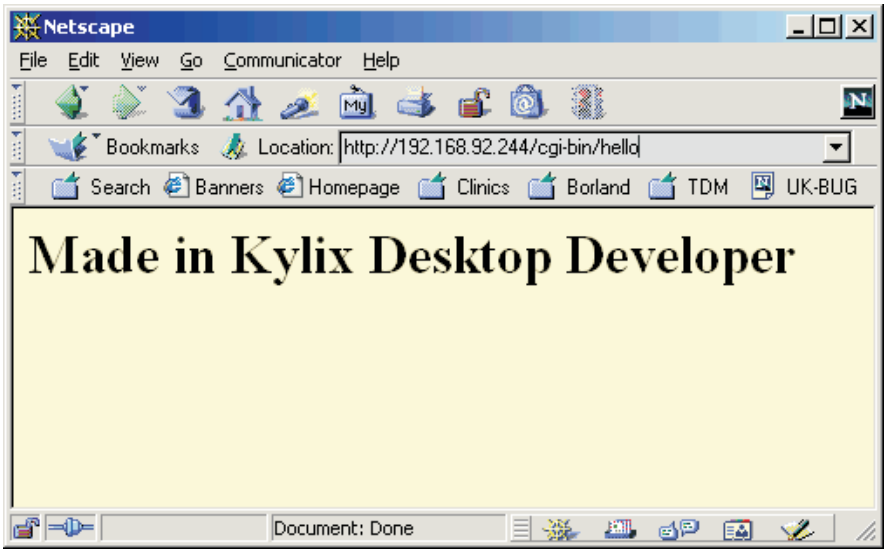
Kylix Deployment

We're almost ready with the first web server application written in Kylix Desktop Developer. 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 *hello* 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 1). Of course, if you are running the application on a remote web server, you need to upload it to your web hosting account's cgi-bin directory.

Die-hard Linux fans who prefer to work with the command line might be happy to know that most of the code in this article can be created using any text editor, after which the dcc command-line Kylix compiler can be used to produce the CGI executable. In this case, we must use the `-E` flag to specify the output directory of the resulting application (still called the EXE output directory, by the way), so my command line is:

```
dcc -E/home/http/cgi-bin
  hello.dpr
```

Remember to make sure that kylix/bin is in your search `PATH` as well as the library `LD_LIBRARY_PATH`, but you can do that by running the command `source kylixpath` (where `kylixpath` is found in your kylix/bin directory).



➤ *Figure 2: Result of Kylix Desktop CGI App (Netscape on Win2000).*

Apache Deployment

We can now compile the *hello* application, and it will indeed result in a *hello* executable file in 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 server applications need. For this, we need to manually edit the httpd.conf file as follows:

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

[For all you vi-haters out there, try the Pico editor, which is a lot friendlier! Ed]. Add a single line to the end of the file, with the following content:

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

After you've modified the httpd.conf file, you need to explicitly restart the Apache web server as follows:

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

Of course, if you are using a remote web server, you need to check with the hosting company that the Kylix libraries are already installed and the path set up.

Now we are ready to start our browser and show the *hello* application. The easiest way for me is to call http://localhost/cgi-bin/hello, or connect to my Linux web server from a browser on another PC to call http://192.168.92.244/cgi-bin/hello and watch the results (see Figure 2).

This concludes the first native Linux web server application written in Kylix Desktop Developer. The nice thing to notice again 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 PC to produce a standard windows CGI application.

User Input

Like I said in the introduction, a CGI application not only produces standard output, but also reads standard input to interact with the end-user (using a browser). Regarding the input, that's not entirely accurate, since a CGI application first has to examine some environment variables (set by the web server when a user request comes in). First of all, we need to determine the value of the `REQUEST_METHOD`, which can be `GET` or `POST`. In the case of the `POST` protocol, the CGI application should then examine the `CONTENT_LENGTH` environment variable to determine the number of characters to read from standard input. This

is the most commonly used REQUEST_METHOD, by the way. The alternative, GET, does not use the standard input, but instead passes the entire request in yet another environment variable called QUERY_STRING. This is slightly faster than reading from standard input, but also limited in the amount of data that you can pass along (I wouldn't try to put more than a few kilobytes inside the QUERY_STRING, for example).

A list of the most important environment variables and their descriptions is shown in Table 1. We've already seen the first three. The HTTP_COOKIE is used to obtain the cookie data for the specific URL. The REMOTE_ADDR is nice if you want to keep track of your visitors (to see if you indeed get different/unique visitors or repeated visits by the same IP-address, which might be the same person). The SCRIPT_NAME returns the URL of the CGI application itself, without the domain, but good enough to call itself again, as we'll do later in this article.

Environment Variable	Description
REQUEST_METHOD	Specifies whether data for the HTTP request was sent as part of the URL (GET) or directly to standard input (POST).
CONTENT_LENGTH	Number of bytes passed to standard input as content from a POST request.
QUERY_STRING	Data passed as part of the URL, comprised of anything after the question in the URL, usually the result of a GET request.
HTTP_COOKIE	A collection of cookies for this specific URL (stored on client machine).
REMOTE_ADDR	End-user's IP address or server name.
HTTP_USER_AGENT	Information about the visitor's web browser.
SCRIPT_NAME	The full name of the script being executed (without the domain part)
PATH_INFO	Additional relative path information passed to the server after the script name, but before any query data, used by WebBroker in Kylix Server Developer.

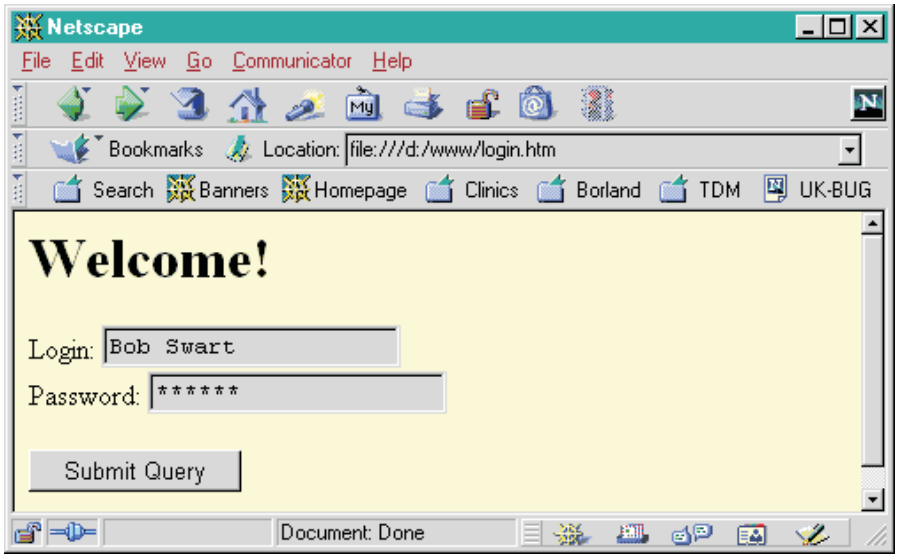
➤ *Table 1*

Finally, the PATH_INFO is used by WebBroker applications to dispatch incoming requests based on the value of this pathinfo. The consequence is that more (different) functionality can be contained in a WebBroker application, than is usually done by several single (but smaller) CGI applications. There are many more environment variables set by the web server, but these are the most relevant in our case today.

➤ *Listing 2: HTML Form login.htm.*

```
<html>
<body bgcolor=ffffcc>
<h1>Welcome!</h1>
<form action="http://192.168.92.244/cgi-bin/login" method=post>
Login: <input type=text name=login>
<br>Password: <input type=password name=password>
<p><input type=submit></p>
</form>
</body>
</html>
```

➤ *Figure 3: HTML CGI Form login.htm (Netscape on Windows NT).*



HTML Form

For the website visitor, who is using a browser to talk to our web server application, we must make an HTML form. This is a bit like a Kylix form, except that the number of different control types is rather limited. Six, to be exact. You can use editboxes, memo fields, listboxes (or drop-down comboboxes), radio buttons, checkboxes and buttons. Buttons are used to fire specific actions, such as reset (everything you typed in the form is lost) or request (send the form input to the server to process it).

As a simple example, let's make an HTML form using two input fields, one named login and another named password (with the type password), as in Listing 2. Save this in a file called login.htm so we can use it in a minute (see Figure 3 for a preview of how it looks in Netscape).

Note that we used METHOD=POST in Listing 2 (so the input fields we send along will not be visible in the URL itself, as they would be if you used METHOD=GET).

Unit DrBobCGI

The big question is: how do I easily examine the environment variables and, after some internal actions, determine the input the user sent to my CGI application? Without WebBroker, which has `ContentFields`, `QueryFIelds` and `CookiesFields`, we're on our own. Fortunately, a few years ago I wrote a unit called `DrBobCGI` for Delphi CGI development on Windows (back in the days when WebBroker was only part of the Client/Server

➤ *Listing 3: DrBobCGI cross-platform for Windows and Linux.*

```
unit drbobcgi;
{=============================================================}
{ unit DrBobCGI © 2001 by Bob Swart (aka Dr.Bob,
    www.drbob42.com }
{ version 1.0 - obtain standard CGI variable values by
   "value()".   }
{ version 2.0 - obtain CGI values, cookies and IP/UserAgent
    values. }
{ version 3.0 - added Linux support for CGI apps with Kylix
    Desktop }
{=============================================================}
interface
type
  TRequestMethod = (Unknown,Get,Post);
var
  RequestMethod: TRequestMethod = Unknown;
var
  ContentLength: Integer = 0;
  RemoteAddress: String[16] = ''; { IP }
  HttpUserAgent: String[128] = ''; { Browser, OS }
  ScriptName: String[128] = ''; { scriptname URL }
  function Value(const Field: ShortString;
    Convert: Boolean = True): ShortString;
  function CookieValue(const Field: ShortString):
    ShortString;
implementation
uses
  {$IFDEF WIN32} Windows, {$ENDIF}
  {$IFDEF LINUX} Libc,    {$ENDIF}
    SysUtils;
function _Value(const Field: ShortString;
  const Data: AnsiString; Sep: Char = '&';
  Convert: Boolean = True): ShortString;
  { 1998/01/02: check for complete match of Field name }
  { 1999/03/01: do conversion *after* searching fields }
  var
    i: Integer;
    Str: String[3];
    len: Byte absolute Result;
  begin
    len := 0; { Result := '' }
    i := Pos('&'+Field+'=',Data);
    if i = 0 then
    begin
      i := Pos(Field+'=',Data);
      if i > 1 then i := 0
    end
    else Inc(i); { skip '&' }
    if i > 0 then
    begin
      Inc(i,Length(Field)+1);
      while Data[i] <> Sep do
      begin
        Inc(len);
        if (Data[i] = '%') and Convert then // special code
        begin
          Str := '$00';
          Str[2] := Data[i+1];
          Str[3] := Data[i+2];
          Inc(i,2);
          Result[len] := Chr(StrToInt(Str))
        end
        else
          if (Data[i] = ' ') and not Convert then
            Result[len] := '+'
          else
            Result[len] := Data[i];
        Inc(i)
      end
    end
    else Result := '$' { no javascript }
  end {_Value};
var
  Data: AnsiString = '';
function Value(const Field: ShortString; Convert: Boolean
    = True): ShortString;
  begin
    Result := _Value(Field, Data, '&', Convert)
  end;
var
  Cookie: AnsiString = '';
function CookieValue(const Field: ShortString):
    ShortString;
  begin
    Result := _Value(Field, Cookie, ';');
    if Result = '' then
      Result := Cookie { debug }
  end;
var
  P: PChar;
  i: Integer;
  Str: ShortString;
initialization
{$IFDEF WIN32}
// Tested on IIS and PWS on Windows
  P := GetEnvironmentStrings;
  while P^ <> #0 do
  begin
    Str := StrPas(P);
    if Pos('REQUEST_METHOD=',Str) > 0 then
    begin
      Delete(Str,1,Pos('=',Str));
      if Str = 'POST' then RequestMethod := Post
      else
        if Str = 'GET' then RequestMethod := Get
    end;
    if Pos('CONTENT_LENGTH=',Str) = 1 then
    begin
      Delete(Str,1,Pos('=',Str));
      ContentLength := StrToInt(Str)
    end;
    if Pos('QUERY_STRING=',Str) > 0 then
    begin
      Delete(Str,1,Pos('=',Str));
      SetLength(Data,Length(Str)+1);
      Data := Str
    end;
    if Pos('HTTP_COOKIE=',Str) > 0 then
    begin
      Delete(Str,1,Pos('=',Str));
      SetLength(Cookie,Length(Str)+1);
      Cookie := Str
    end
    else
    if Pos('REMOTE_ADDR',Str) = 1 then // TDM #39
    begin
      Delete(Str,1,Pos('=',Str));
      RemoteAddress := Str
    end
    else
    if Pos('HTTP_USER_AGENT',Str) = 1 then // TDM #39
    begin
      Delete(Str,1,Pos('=',Str));
      if Pos(')',Str) > 0 then
        Delete(Str,Pos(')',Str)+1,Length(Str)); {!!}
      HttpUserAgent := Str
    end
    else
    if Pos('SCRIPT_NAME',Str) = 1 then // TDM #71
    begin
      Delete(Str,1,Pos('=',Str));
      ScriptName := Str
    end;
    Inc(P, StrLen(P)+1)
  end;
{$ENDIF}
{$IFDEF LINUX}
// Tested on Apache for Linux
  P := getenv('REQUEST_METHOD');
  if P = 'POST' then RequestMethod := Post
  else
    if P = 'GET' then RequestMethod := Get;
  ContentLength := StrToIntDef(getenv('CONTENT_LENGTH'),0);
  Data := getenv('QUERY_STRING');
  Cookie := StrPas(getenv('HTTP_COOKIE'));
  RemoteAddress := StrPas(getenv('REMOTE_ADDR'));
  HttpUserAgent := StrPas(getenv('HTTP_USER_AGENT'));
  ScriptName := StrPas(getenv('SCRIPT_NAME'));
{$ENDIF}
// single source cross-platform from now on
  if RequestMethod = Post then
  begin
    SetLength(Data,ContentLength+1);
    for i:=1 to ContentLength do read(Data[i]);
    Data[ContentLength+1] := '&';
  { if IOResult <> 0 then { skip } }
  end;
  i := 0;
  while i < Length(Data) do
  begin
    Inc(i);
    if Data[i] = '+' then Data[i] := ' '
  end;
  if i > 0 then Data[i+1] := '&'
    else Data := '&';
finalization
  Data := ''
end.
```

edition of Delphi: it almost feels like history is repeating itself, and Borland needs to re-learn a lesson about marketing WebBroker!). Anyway, this unit DrBobCGI did all the dirty low-level work for us, and we only had to call a single function called Value to get the value of a CGI variable (like login or password if you use the form in Listing 2), independent of the request methods used (GET or POST). I've even added a function called CookieValue to get the same functionality for cookies.

Furthermore, we have a few global variables that are set to the values of REMOTE_ADDR (the IP address of the visitor and the browser) and the HTTP_USER_AGENT which contains information about the browser itself. This way, you can not only keep track of the different visitors, but also of the different kinds of browsers (and operating systems). See *The Delphi Magazine* Issue 39 for more details about these features, which may not be required for your tasks. Still, it's all contained in DrBobCGI which is available on this month's disk (as well as my own website at www.drbob42.com/tools) for you to use, but always at your own risk, of course.

And the good thing is that I have now made it cross-platform by using $IFDEFs, so you can now use DrBobCGI with Kylix as well as Delphi (see Listing 3). The main differences are the way in which we can access environment variables in Windows (where we get all the name=value pairs using a single GetEnvironmentStrings) and Linux (where we have to use getenv for every environment variable name we want to know the value of).

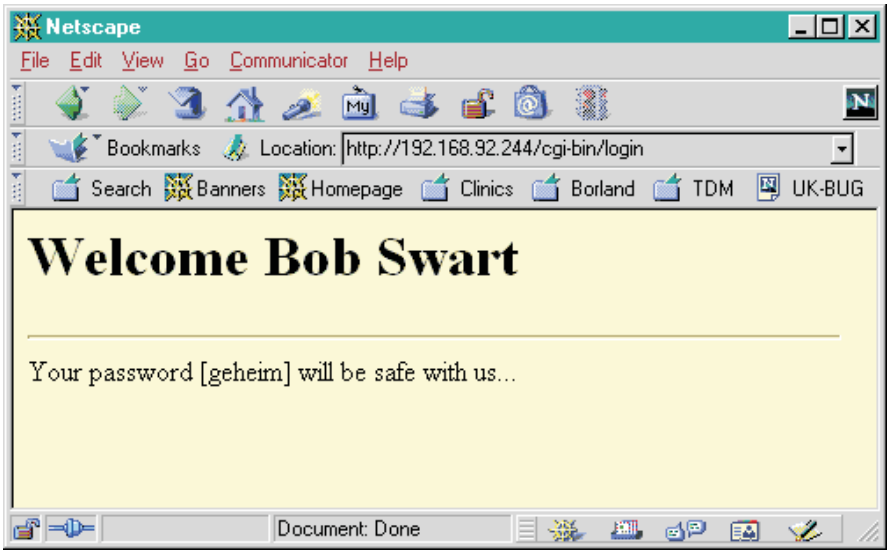
CGI Login Example

The only thing that's left now is writing a CGI application to 'answer' the login.htm form, and respond to the login and password information being passed on (for that we can use the DrBobCGI unit). This turns out to be very easy as well, as Listing 4 demonstrates.

Figure 4 shows the output of the CGI application login running on Linux. And as I've said before, this could have been a Windows CGI application just as easily. The only way by which you can tell that you're looking at a Linux hosted CGI application and not a Windows hosted one is the extension: there is none. The full URL is http://192.168.92.244/cgi-bin/login (on Windows it would be login.exe). Of course, you need to make sure that the HTML form also uses the correct call (ie never use .exe or .dll extensions on Linux).

```
program login;
{$APPTYPE CONSOLE}
uses
  DrBobCGI;
begin
  writeln('content-type: text/html');
  writeln;
  writeln('<html>');
  writeln('<body bgcolor=ffffcc>');
  writeln('<h1>Welcome ' + Value('login') + '</h1>');
  writeln('<hr>');
  writeln('Your password [', Value('password'), '] will be safe with us...');
  writeln('</body>');
  writeln('</html>')
end.
```

➤ *Listing 4: Program login.dpr (which runs on Windows just as well as Linux).*



➤ *Figure 4: CGI application login result (Netscape on WinNT).*

DrBobCGI Additional

Now, let's see if the other features inside DrBobCGI also work on Linux. These include support for cookies (using the HTTP_COOKIE environment variable, which makes an excellent follow-up story for my website at www.drbob42.com/examines) as well as the HTTP_USER_AGENT and REMOTE_ADDR environment variables. The latter two are easy to use: just write the values of the variables RemoteAddress and HttpUserAgent from the DrBobCGI unit (we'll get back to the SCRIPT_NAME later in this article):

```
writeln('RemoteAddr: ',
  RemoteAddress);
writeln('UserAgent: ',
  HttpUserAgent);
```

The result is not unexpected: some additional information about the client's machine (see Figure 5).

dbExpress Example

After all this regular HTML producing, it's time to do something a little bit more interesting. How about connecting to a dbExpress dataset, browsing through the records inside? Or presenting them in a grid-like overview? Both take just a second when using WebBroker, because you can use the DataSetPageProducer and DataSetTableProducer components. However, even without these components, you only need

a little bit of HTML knowledge to produce the output you want.

Before we start, please do make sure you've read Brian Long's excellent article in the May issue of *The Delphi Magazine* on configuring Kylix and Apache (especially the section on making it work with dbExpress). Basically, you need to make sure to have done a `SetEnv HOME` to a directory that holds a hidden .borland directory with your dbExpress configuration files.

InterBase On The Web

Anyway, to start with an easy example, let's use dbExpress to connect to the `IBLocal` InterBase database and show the contents of the `CUSTOMER` table. If you don't know the names and types of the fields, you can still write generic code that will take any `TDataSet` or derived component and produce a full HTML file with the contents of this dataset. For those of you with little HTML knowledge: an HTML table starts with `<table>` and ends with `</table>`. A table row (containing a dataset record) starts with `<tr>` and ends with `</tr>` while a table cell (a field of a record) starts with `<td>` and ends with `</td>`. I want to start the table with a row that only contains the fieldnames, and then follow up with a row for each record in the table, showing the values of each field of each record. The generic code, which again works on any `TDataSet`, can be see in Listing 5.

The next step is determining which table or query to work with. If you don't want to see all the fields, just make sure you don't select them all. Of course, the routine in Listing 5 can be modified to your needs. I always find a generic routine to be a good starting point.

```
procedure DataSet2HTML(const DataSet: TDataSet);
var
  fields: Integer;
begin
  writeln('<table border=1>');
  DataSet.Open;
  write('<tr>');
  for fields:=0 to Pred(DataSet.FieldCount) do
    write('<td bgcolor=ffffff><b>',DataSet.Fields[fields].FieldName,'</td>');
  writeln('</tr>');
  DataSet.First;
  while not DataSet.Eof do begin
    write('<tr>');
    for fields:=0 to Pred(DataSet.FieldCount) do
      write('<td>',DataSet.Fields[fields].AsString,'</td>');
    writeln('</tr>');
    DataSet.Next
  end;
  writeln('</table>')
end {DataSet2HTML};
```

➤ *Listing 5: DataSet 2 HTML conversion (cross-platform code).*

An example CGI application that just opens the entire customer table from the `IBLocal` database, using a `SQLDataSet` with the SQL command `select * from customer` is shown in Listing 6. The `SQLDataSet` is passed as parameter to the `DataSet2HTML` function, where the dynamic HTML grid is generated.

The code in Listing 6 only compiles with CLX (not with the VCL), which means Kylix on Linux (or Delphi 6: although I haven't received the shipping version yet I assume it's OK). The output generated by this simple application can be seen in Figure 6.

This is truly the result of SQL as it's supposed to be: perform a query, get the results and display them. And for that, we only need a unidirectional cursor, one that dbExpress provides, as we saw in previous months.

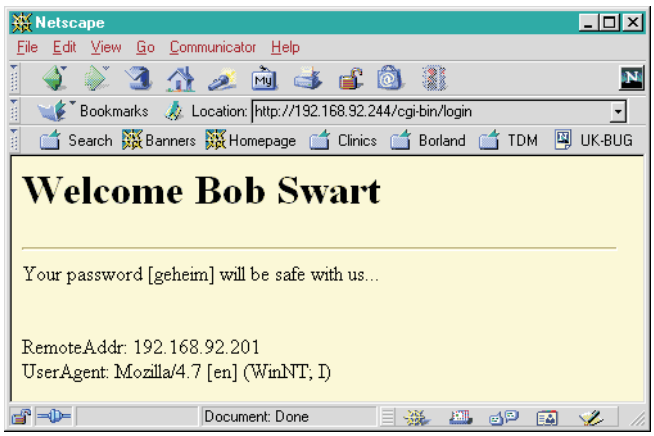
However, we're not done yet. First of all, the code from Listing 6 should be made more reusable by turning it into a routine where we can pass the `ConnectionName` as well as the SQL string. That would make it a more valuable snippet. The final example illustrates this, and involves maintaining state as well!

More dbExpress

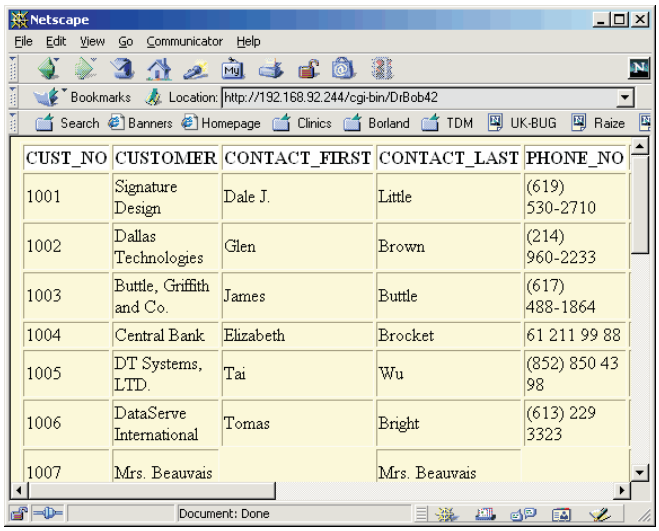
Another example that I want to share with you shows how to browse through the result set and view one record at a time. Browse through a unidirectional dataset? Yes, because we only request one record at a time, we can safely return and view the previous record next time.

But first things first: let's start with producing dynamic HTML for a single record from an open dataset. This code can be seen in

➤ *Figure 5: CGI application login result (Netscape on WinNT).*



➤ *Figure 6: CGI application DrBob42 result (Netscape on Win2000).*



Listing 7 (compare this to Listing 5, which generates HTML for the entire DataSet instead of a single record).

Note that I'm using the AsString value of each field here, but did not do anything special for images. Like I said, that will be handled in a follow-up article.

Navigating In HTML

The interesting part is navigating a unidirectional dataset. Is this a problem, or not? In fact it's not a problem. A unidirectional dataset can only move forward (one step) or reset itself to the first record, but remember that our CGI application only shows one record at a time. And every time it's called, it will again open the same unidirectional dataset, and position itself to the correct record (which potentially could be the one prior to the record we just saw). As a result, we can scroll back, although we are re-opening the unidirectional dataset everytime.

The technique that we'll use in this example is based on the fact that the CGI application itself will be called with an optional GET parameter that specifies the record number to display. The GET parameter, listed in the requesting URL itself, is called RecNo, and has the value 1 for the first record, 2 for the second record, and so on. If we want to jump to the last record, we have a slight problem, because we don't know beforehand how many records are in the dataset. As a workaround, I'm passing -1 to indicate that I want to go to and show the last record (this still involves two passes when we need to position the dataset, as we'll see in a moment).

The biggest trick, however, is that the CGI application needs to call itself. Fortunately, we've already seen that the ScriptName has the exact value of the CGI application (/cgi-bin/DrBob42 in this case), so we can simply use ScriptName, and never have to worry when we move the CGI application to another machine (which would have been the case if we had hardcoded a domain or IP-address here).

```
var
  DataSet: TSQLDataSet;
  SQLConnection1: TSQLConnection;
begin
  writeln('content-type: text/html');
  writeln;
  writeln('<html>');
  writeln('<body bgcolor=ffffcc>');
  SQLConnection1 := TSQLConnection.Create(nil);
  with SQLConnection1 do
  begin
    LoadParamsOnConnect := True;
    ConnectionName := 'IBLocal';
    LoginPrompt := False;
    Connected := True;
  end;
  DataSet := TSQLDataSet.Create(nil);
  try
    DataSet.SQLConnection := SQLConnection1;
    DataSet.CommandText := 'select * from customer';
    DataSet2HTML(DataSet);
  finally
    DataSet.Free;
    SQLConnection1.Free
  end;
  writeln('</body>');
  writeln('</html>')
end.
```

➤ *Listing 6: Connecting to IBLocal InterBase database table customer (CLX code).*

```
procedure Record2HTML(const DataSet: TDataSet);
var
  fields: Integer;
begin
  if not DataSet.Active then DataSet.Open;
  for fields:=0 to Pred(DataSet.FieldCount) do
    writeln('<b>',DataSet.Fields[fields].FieldName,':</b> ',
      DataSet.Fields[fields].AsString,'<br>')
end {Record2HTML};
```

➤ *Listing 7: Record 2 HTML conversion (cross-platform code).*

```
procedure NavigatorHTML(const DataSet: TDataSet; RecNo: Integer);
begin
  if RecNo = 0 then RecNo := 1;
  if not DataSet.Active then DataSet.Open;
  write('<a href="',ScriptName,'?RecNo=1">First</a> | ');
  write('<a href="',ScriptName,'?RecNo=',Pred(RecNo),'">Prior</a> | ');
  write('<a href="',ScriptName,'?RecNo=',Succ(RecNo),'">Next</a> | ');
  write('<a href="',ScriptName,'?RecNo=-1">Last</a> | ');
  write('<a href="',ScriptName,'?RecNo=',RecNo,'">Refresh</a> (',RecNo,')<br>')
end {NavigatorHTML};
```

➤ *Listing 8: Navigator HTML conversion (cross-platform code).*

Positions Everyone?

Now that we have a Record2HTML and a NavigatorHTML it's time to put the pieces together. This time I will also make sure to parameterise the whole thing so we can effectively reuse all our code so far (it may not be components like WebBroker, but it certainly beats reinventing the wheel every time).

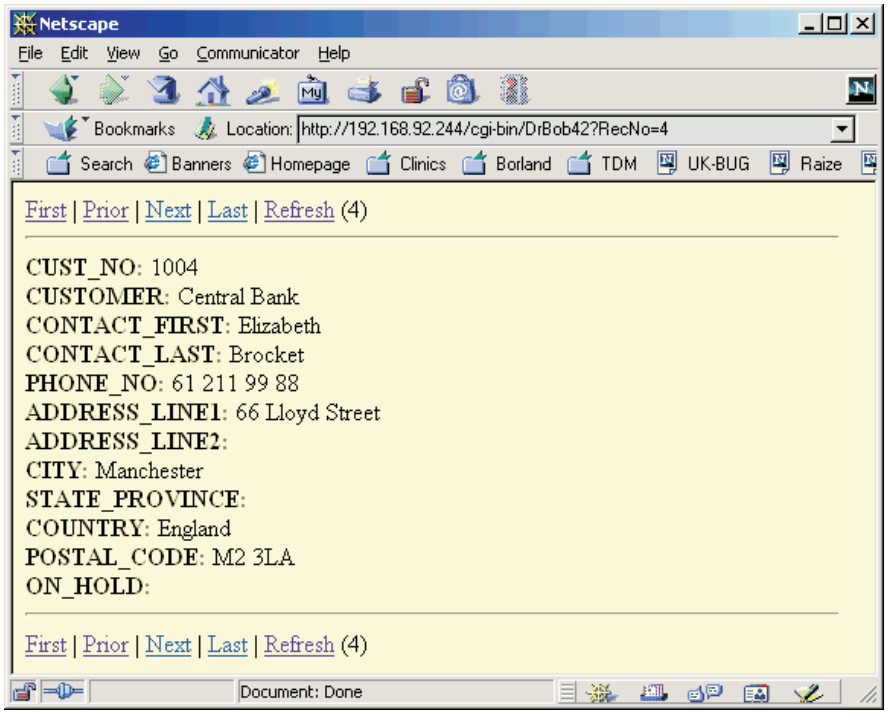
The big routine will be called DBQuery2HTML, taking two arguments: DB and Query. The first argument specifies the SQLConnection name to connect to (like IBLocal), while the second argument contains the entire SQL query you want to perform. Once we've created a connection and dataset and opened the dataset, it's time to use the Value routine from DrBobCGI to see if a RecNo field was passed on the URL (using the GET protocol). If not, then we can assume that we've just started the CGI application for the first time, so RecNo should be set to 1. The SysUtils unit contains a useful routine that we can use for this purpose, namely StrToIntDef. This attempts to convert the string value returned by Value('RecNo') and if it fails for some reason, returns the second argument (the Def for default) as an integer value. Internally this is probably done by a try..except block, but it saves me from doing the same in my

```
procedure DBQuery2HTML(const DB, Query: String);
var
  DataSet: TSQLDataSet;
  SQLConnection1: TSQLConnection;
  RecNo,i: Integer;
begin
  SQLConnection1 := TSQLConnection.Create(nil);
  with SQLConnection1 do
  begin
    LoadParamsOnConnect := True;
    ConnectionName := DB;
    LoginPrompt := False;
    Connected := True;
  end;
  DataSet := TSQLDataSet.Create(nil);
  try
    DataSet.SQLConnection := SQLConnection1;
    DataSet.CommandText := Query;
    DataSet.Open;
    RecNo := StrToIntDef(Value('RecNo'),1);
    if RecNo = -1 then
    begin
      RecNo := 1;
      while not DataSet.Eof do
      begin
        Inc(RecNo);
        DataSet.Next
      end
    end
    else
      for i:=1 to Pred(RecNo) do DataSet.Next;
    if DataSet.Eof then // went past Eof, need to backtrack!
    begin
      Dec(RecNo); // one before Eof
      DataSet.First;
      for i:=1 to Pred(RecNo) do DataSet.Next
    end;
    // DataSet2HTML(DataSet);
    NavigatorHTML(DataSet,RecNo);
    writeln('<hr>');
    Record2HTML(DataSet);
    writeln('<hr>');
    NavigatorHTML(DataSet,RecNo);
  finally
    DataSet.Free;
    SQLConnection1.Free
  end;
end {DBQuery2HTML};
```

➤ *Listing 9: DBQuery 2 HTML conversion (cross-platform code).*

```
begin
  writeln('content-type: text/html');
  writeln;
  writeln('<html>');
  writeln('<body bgcolor=ffffcc>');
  DBQuery2HTML('IBLocal','select * from customer');
  writeln('</body>');
  writeln('</html>')
end.
```

➤ *Listing 10: Performing a query to a IBLocal InterBase customer table (CLX code).*



➤ *Figure 7: CGI application DrBob42 result.*

code (resulting in less code, easier to read).

The value of `RecNo` can be anything from 1 to a high number. As a special meaning, it can also be -1, because we passed that value to go to the last record. The problem that we have when working with unidirectional datasets is that you cannot just walk until you reach `DataSet.Eof`, because at that time the last record will be gone already. You need to stop just before that. I've solved this by using a two-phase technique: first I walk through the dataset, counting the number of records until `DataSet.Eof`. Then, in the next pass, I start from the beginning again but this time make sure to move the number of records minus one (so I end at the last record).

Note that the same second pass is used when the user advances to the next record, which also results in `DataSet.Eof` to be true. Again, this means that the user went one record too far, so the application should reset the dataset and walk it again, this time one record less far.

When the dataset is positioned correctly, we can finally call `NavigatorHTML`, passing the positioned dataset as well as the `RecNo` (which is needed to specify the previous and next values or `RecNo`), followed by the call to `Record2HTML` to display the fields and field values of the current record.

Wrap Up!

Now that we have a `DBQuery2HTML` routine that calls `Record2HTML` as well as `NavigatorHTML` (and potentially also `DataSet2HTML`), the only thing we need is a main CGI application that sets the content-type, outputs the `<html>` and `<body>` elements, and calls `DBQuery2HTML` with the correct `ConnectionName` and `Query` SQL statement.

Note that this solution will be usable with Kylix as well as with Delphi 6, and you can use it with IBLocal as well as any other dbExpress database that you have available (although I must confess I have only tested it against InterBase on Linux so far: if you experience any problems, just let

me know and we'll see how we can solve them).

Listing 10 contains the main application, in our case performing the query `select * from customer` on the `IBLocal` InterBase database (on Linux). Figure 6 shows the final output.

There are a number of possible enhancements we can make. For example, when converting a table that holds images (or other binary BLOB data), we may want to write a 'helper' CGI application that can be used to generate the dynamic image for that particular record, while the field value for that record should be turned into an indirect call to this helper application. I'll cover this in a follow-up article.

Next Time

In the past few months, we've examined web server development using Kylix, WebBroker in Kylix Server Developer and 'plain' `DrBobCGI` and more in Kylix Desktop Developer.

In the meantime, however, Delphi 6 has been launched and, by the time you read this article, you may have heard a lot about the new features called WebSnap and BizSnap, available in Delphi 6 Enterprise. These will be the topics for the next few months, and I promise you that web development with Delphi will never be the same again. *So stay tuned...*

Bob Swart (aka Dr.Bob, www.drbob42.com) is an @-Consultant, Delphi Clinic Trainer and co-founder of the Kylix/Delphi OplossingsCentrum (see www.KDOC.nl) in The Netherlands.