Under Construction:
A Query HTML CGI-Form Wizard

by Bob Swart

This month we'll develop yet another variant of TableBob, my 'table to source code or HTML' wizard, to generate an HTML CGI-Form that can be used to enhance the Delphi 3 Web Modules. Techniques that we'll use along the way include wizards/experts, component editors and property editors, plus a little internet programming.

Although this article ends with an enhancement for Delphi 3's Web Modules, the resulting Query2CGI wizard is also of great value without Web Modules, as will become clear shortly. However, for now, I assume that you will know the basics of Delphi 3 Web Modules (if not, check Issues 24 and 25).

CGI-Forming

Back in Issue 25, I noted that it was rather inconvenient that Delphi supports all kinds of protocols with the Web Modules, but we still have to write our own HTML files to hold the CGI forms. This is especially irritating when using a `TQueryTableProducer` together with a Parameterised Query (where we can automatically 'connect' a CGI variable in the HTML page with a Query parameter if they use the same name). I used to have an HTML CGI form 'template' lying around, that I could customise. But I felt I was missing something. So I decided to build my own Query to HTML CGI form wizard.

```
procedure TFormWizard.ButtonStepClick(Sender: TObject);
begin
  if Sender IS TButton then
    NoteBook.PageIndex := NoteBook.PageIndex + (Sender AS TButton).Tag;
  ButtonBack.Enabled := NoteBook.PageIndex > 0; { first }
  if (NoteBook.PageIndex = 1) and ((Sender AS TButton).Tag = 1) then
    BuildSQL;
  Caption := Title + Format(' %d/%d', [NoteBook.PageIndex+1,
    NoteBook.Pages.Count]);
  if NoteBook.PageIndex < Pred(NoteBook.Pages.Count) then begin
    ButtonNext.Caption := '&Next >';
    ButtonNext.ModalResult := mrNone
  end else begin
    { Finish }
    ButtonNext.Caption := '&Finish';
    ButtonNext.ModalResult := mrOk
  end
end;
```

➤ *Listing 1: ButtonStepClick*

Query-2-HTML

In order to write our wizard, we must first find out what data we actually need. Of course, we need the Query's SQL statement (and probably the `DatabaseName` or `Alias` as well). Next, we need some way to specify which fields of the result set must end up in the generated HTML file. Finally, since we're creating an HTML CGI form, we also need to specify the `Action` and `Method` of the `FORM` itself. These three steps will be the starting point for a wizard, using the `TFormWizard` (see Issue 21) as a template.

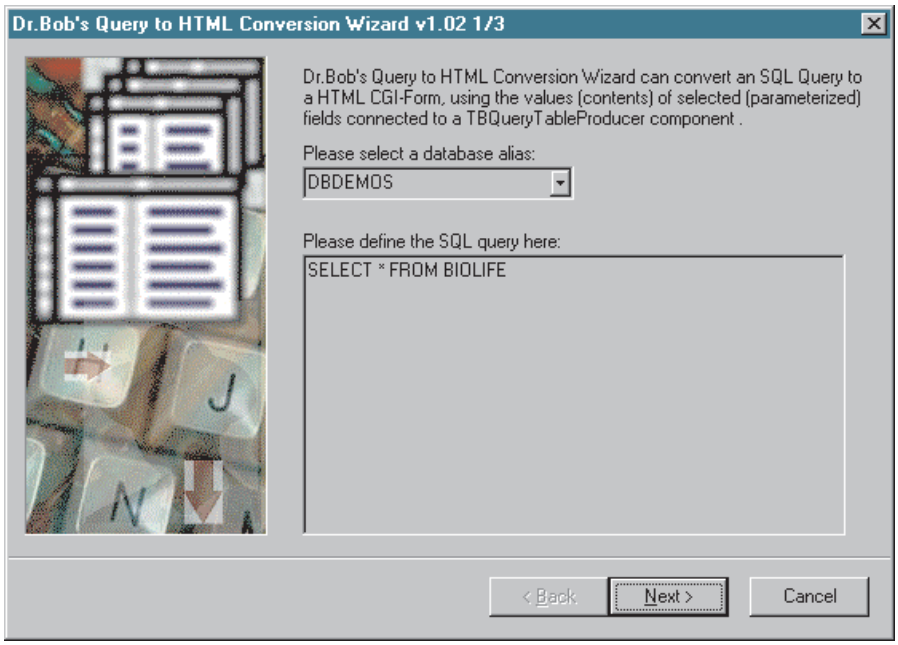
When the first page of the wizard is created we need to put a list of all known BDE Database Aliases in the drop-down combobox, by calling the `Session.GetAliasNames` routine in the `FormCreate` event:

```
procedure TFormWizard.FormCreate(
  Sender: TObject);
begin
  Session.GetAliasNames(
    ComboBoxAliases.Items)
end;
```

Both the `Next` and the `Prev` button are connected to the same event: `ButtonStepClick` (Listing 1). The only difference is that the `Next` button has a `Tag` value of `1`, and the `Prev` button has a `Tag` value of `-1`. The `Tag` value defines the 'steps' to take, forwards or backwards.

If we click on the `Next` button for the first time, we go to the second page of the `NoteBook` (ie `NoteBook.PageIndex = 1`), which means

➤ *Figure 1*



that the SQL Query should now be executed and a list of field names can be presented in a checklistbox (so we can specify which fields are relevant): see Figure 2.

The source code of `BuildSQL` itself, which extracts the available fieldnames from the Query without actually *executing* the Query itself, is shown in Listing 2. Note that we also make sure that only parameterised query fields (using a colon prefix) are pre-selected in the checklist.

Unfortunately, checking for parameterised fields in the Query's SQL statement is of no use at this time, since we can't execute a Parameterised Query in the above way. We need to specify the type of each Parameter, or else we get an exception. We'll remedy this problem later, when we turn to Component and Property Editors.

The final page in the wizard is used to specify some CGI details, such as the `Action` and `Method`, as well as the `Title` (caption) we want the HTML page to have and the file to save the form in (Figure 3).

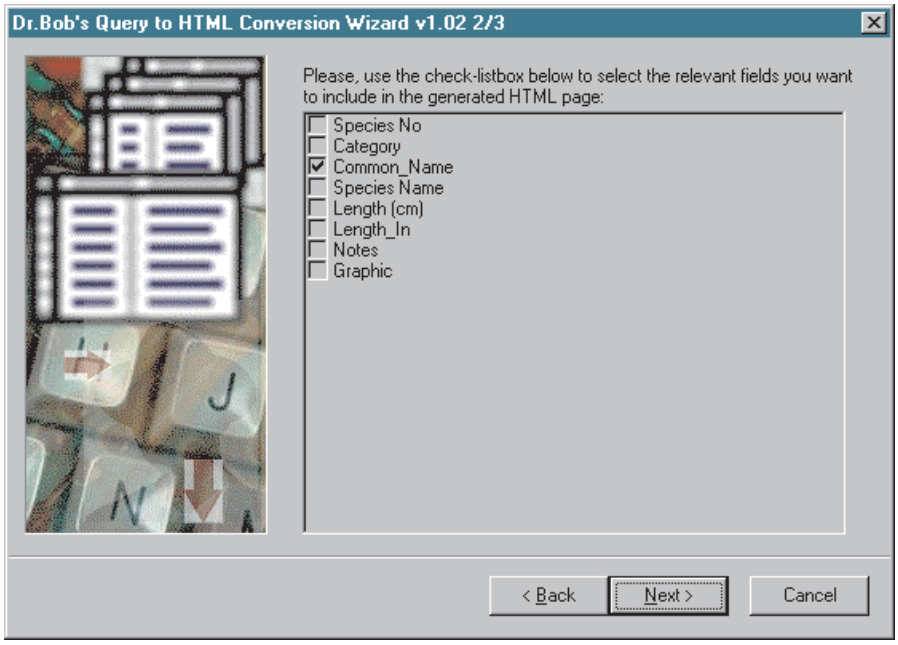
Once we've collected all information from the three wizard pages, the procedure `Finish` is relatively straightforward. We need to create an HTML file with a CGI form inside, using `<FORM ACTION=... METHOD=...>` and `</FORM>` tags. For each selected fieldname of the Query, we need to put every possible value from that field (as a Query result) in a drop-down ComboBox, using `<SELECT>` and `<OPTION>` tags. The HTML details are not important right now. See Listing 3.

And that's basically all there is to it. The end result (unit `DrBobWiz`) contains the building blocks for the remainder of the article.

Stand-Alone?

The first thing to do is test the new Query-2-HTML wizard as a stand-alone executable (Listing 4). This sure sounds like a useful utility to have and Figures 1 to 3 were all generated using this version.

Note that we don't call procedure `Finish` if the user simply closes the wizard or clicks on the `Cancel` button (which results in a `ModalResult` of `mrCancel`).

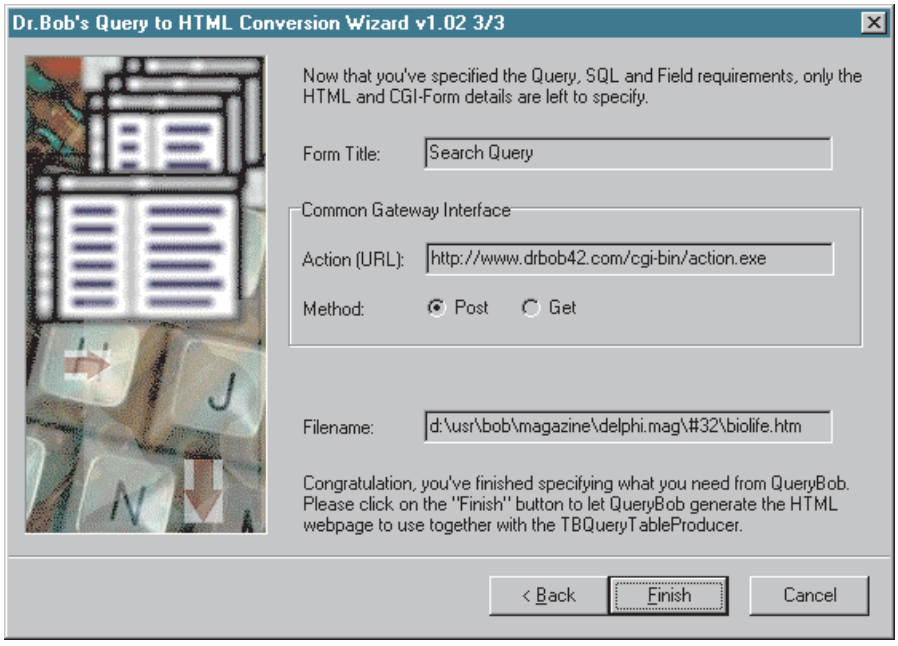


➤ *Figure 2*

```
procedure TFormWizard.BuildSQL;
var
  i: Integer;
begin
  CheckListFields.Items.Clear;
  with TheQuery do begin
    if ComboBoxAliases.Text <> '' then
      DatabaseName := ComboBoxAliases.Text;
    SQL := MemoSQL.Lines;
    FieldDefs.Update { get info without executing TheQuery };
    for i:=0 to Pred(FieldDefs.Count) do begin
      CheckListFields.Items.Add(FieldDefs[i].Name);
      { only select the "parameterised" fields in the Query }
      CheckListFields.Checked[i] :=
        (Pos(UpperCase(':'+FieldDefs[i].Name),UpperCase(SQL.Text)) > 0) or
        (Pos(UpperCase(':"'+FieldDefs[i].Name+'"'),UpperCase(SQL.Text)) > 0);
    end
  end
end {BuildSQL};
```

➤ *Listing 2: BuildSQL*

➤ *Figure 3*



```
procedure TFormWizard.Finish;
var
  f: System.Text;
  Str: String;
  i: Integer;
begin
  begin
    System.Assign(f,EditFileName.Text);
    Rewrite(f);
    writeln(f,'<HTML>');
    writeln(f,'<BODY>');
    writeln(f,'<H1>',EditTitle.Text,'</H1>');
    writeln(f,'<HR>');
    write(f,'<FORM ACTION="',EditAction.Text);
    if RadioPost.Checked then
      writeln(f,'" METHOD=POST>')
    else
      writeln(f,'" METHOD=GET>');
    writeln(f,'<UL>');
    with TheQuery do
    try
      if ComboBoxAliases.Text <> '' then
        DatabaseName := ComboBoxAliases.Text;
      Str := MemoSQL.Text;
      { ignore "WHERE" part of Query; generate all values }
      if Pos('WHERE ',UpperCase(Str)) > 0 then
        System.Delete(Str, Pos('WHERE ', UpperCase(Str)),
          Length(Str));
      SQL.Text := Str;
      Open;
      for i:=0 to Pred(CheckListFields.Items.Count) do
        if CheckListFields.Checked[i] then begin
          writeln(f,'<LI>',CheckListFields.Items[i],':');
          writeln(f,'<BR><SELECT NAME="',
            CheckListFields.Items[i],'">');
          First;
          while not Eof do begin
            writeln(f,'<OPTION VALUE="', FieldByName(
              CheckListFields.Items[i]).AsString, '"> ',
              FieldByName(
                CheckListFields.Items[i]).AsString);
            Next
          end;
          writeln(f,'</SELECT>');
          writeln(f,'<P>')
        end;
        writeln(f,'</UL>');
        writeln(f,'<CENTER>');
        writeln(f,'<INPUT TYPE=RESET>');
        writeln(f,'<INPUT TYPE=SUBMIT>');
        writeln(f,'</FORM>');
        writeln(f,'<HR>');
        writeln(f, 'Generated by QueryBob (c) 1998 '+
          'by Bob Swart (aka Dr.Bob)');
        writeln(f,'</BODY>');
        writeln(f,'</HTML>');
        Close
    finally
      System.Close(f);
    end
  end
end {Finish};
```

➤ *Listing 3: Finish*

For the session that resulted in Figures 1 to 3, the results are shown in Listing 5: an HTML form listing values of the `Common_Name` field of the `BIOLIFE.DB` table . When connected to a `TQueryTableProducer`, this could be used for a web application where one could easily select the `Common_Name` for which more details should be returned.

Note that the items in the drop-down ComboBox are not sorted (Figure 4). This could easily be fixed by adding `ORDER BY Common_Name` to the SQL Query that we defined in Figure 1.

Expert/Wizard?

While it's certainly nice to have our wizard as a stand-alone application, let's now focus on ways to actually integrate it into the Delphi IDE, so we can offer even more help at design-time. Usually, the first approach I use is to write a Delphi IDE wizard 'wrapper' around it (generated by my Wizard wizard, see Issue 21).

For example, to create an AddIn wizard that is inserted in the `Database` menu after the `Database Form Wizard`, I need to find the `Borland_FormExpertMenu` menu item, get to its parent menu and insert my own menu item right after the index of the former: see Listing 6.

In the `OnClick` event handler I call the method `Execute` (which is also used by the other three wizard

```
program Test;
uses
  Forms, Controls,
  DrBobWiz in 'DrBobWiz.pas' {FormWizard};
begin
  Application.Initialize;
  with TFormWizard.Create(Application) do
  try
    if ShowModal = mrOK then Finish
  finally
    Free
  end
end.
```

➤ *Listing 4: Stand-Alone Test*

```
<HTML>
<BODY>
<H1>Search Query</H1>
<HR>
<FORM ACTION="http://www.drbob42.com/cgi-bin/action.exe" METHOD=POST>
<UL>
<LI>Common_Name:
<BR><SELECT NAME="Common_Name">
<OPTION VALUE="Clown Triggerfish"> Clown Triggerfish
<OPTION VALUE="Red Emperor"> Red Emperor
<OPTION VALUE="Giant Maori Wrasse"> Giant Maori Wrasse
<OPTION VALUE="Blue Angelfish"> Blue Angelfish
<OPTION VALUE="Lunartail Rockcod"> Lunartail Rockcod
<OPTION VALUE="Firefish"> Firefish
<OPTION VALUE="Ornate Butterflyfish"> Ornate Butterflyfish
<OPTION VALUE="Swell Shark"> Swell Shark
<OPTION VALUE="Bat Ray"> Bat Ray
<OPTION VALUE="California Moray"> California Moray
<OPTION VALUE="Lingcod"> Lingcod
<OPTION VALUE="Cabezon"> Cabezon
<OPTION VALUE="Atlantic Spadefish"> Atlantic Spadefish
<OPTION VALUE="Nurse Shark"> Nurse Shark
<OPTION VALUE="Spotted Eagle Ray"> Spotted Eagle Ray
<OPTION VALUE="Yellowtail Snapper"> Yellowtail Snapper
<OPTION VALUE="Redband Parrotfish"> Redband Parrotfish
<OPTION VALUE="Great Barracuda"> Great Barracuda
<OPTION VALUE="French Grunt"> French Grunt
<OPTION VALUE="Dog Snapper"> Dog Snapper
<OPTION VALUE="Nassau Grouper"> Nassau Grouper
<OPTION VALUE="Bluehead Wrasse"> Bluehead Wrasse
<OPTION VALUE="Yellow Jack"> Yellow Jack
<OPTION VALUE="Redtail Surfperch"> Redtail Surfperch
<OPTION VALUE="White Sea Bass"> White Sea Bass
<OPTION VALUE="Rock Greenling"> Rock Greenling
<OPTION VALUE="Senorita"> Senorita
<OPTION VALUE="Surf Smelt"> Surf Smelt
</SELECT>
</UL>
<P>
<CENTER><INPUT TYPE=RESET><INPUT TYPE=SUBMIT>
</FORM>
<HR>
<FONT SIZE=1>Generated by QueryBob (c) 1998 by Bob Swart (aka Dr.Bob)</FONT>
</BODY>
</HTML>
```

➤ *Listing 5: BIOLIFE.HTM (output)*

types), and inside Execute I basically do the same as I did in the stand-alone version: create the TFormWizard, perform a ShowModal and call Finish when needed: see Listing 7.

When registered and installed into a Delphi 3 package, this TBAddInWizard successfully integrates the wizard into the Delphi IDE (right after the Table Form Wizard in the Database menu). However, I do feel that we cannot really speak about true integration here, as there's no connection to a TQueryTableProducer or even a TQuery component. While the wizard is now started from the IDE, it's still as isolated from our web modules project as before.

Component Editor?

Component editors, like wizards, are used to enhance the Delphi IDE. They are also derived from a single base class where some abstract methods need to be overridden and re-defined in order to give the component editor the desired behaviour. Component editors are always bound to a particular component type (see Issue 8 for more information).

Ideally, I would like to show the wizard as a component editor action for the TQueryTableProducer: ie when I click with the right mouse button on a TQueryTableProducer, I would like to get a pop-up menu that lists this wizard, among the other actions that exist for a TQueryTableProducer (such as the Action Editor, for example).

The definition for a TQueryTableProducerComponentEditor, derived from the base class TComponentEditor (defined in DSGNINTF.PAS) is shown in Listing 8.

GetVerbCount returns the number of pop-up menu items for this component editor, which should be 1 (our wizard), which is the result of a call to GetVerb. Like GetVerb, the ExecuteVerb method can safely ignore the Index argument, since we only have one Verb defined: see Listing 9.

Note that this time we're able to extract actual information from the TQueryTableProducer to 'initialise' the Query on the wizard. For example, we can access the value of the QueryDatabaseName, the Query.SQL and even the Query.Params (which is useful for parameterised queries, for which we previously could not offer any support).

In fact, once we have a Web Module project with a Query connected to a TQueryTableProducer, we can use the Component of our component editor (which is the TQueryTableProducer) to access its Query and just grab anything we need, without having to fill this information in (again) by ourselves.

So, ideally, we would start the component editor and just click on the Next button twice, then provide the form information on the last page of the wizard and the Finish action. Even the 'parameterised fields' would automatically have been selected, as we already showed in Listing 2 some time ago.

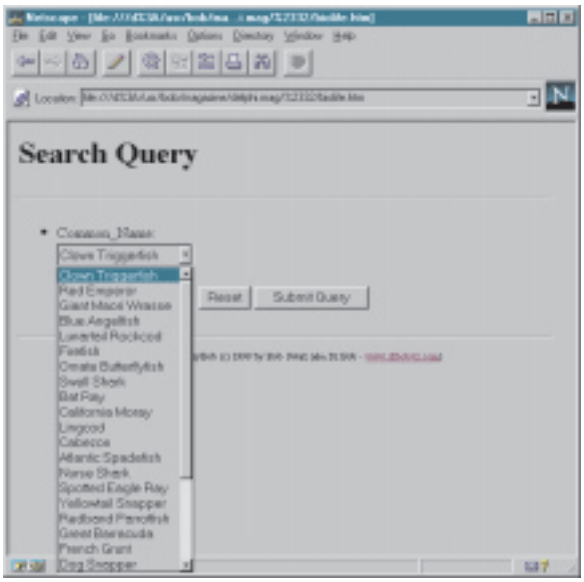
There is one *big* problem with using the TQueryTableProducerComponentEditor, however: once we install it, we effectively 'hide' the previous component editor for TQueryTableProducer, so we no

```
constructor TBAddInWizard.Create;
var
  MainMenu: TIMainMenuIntf;
  MainItem: TIMenuItemIntf;
  MenuItem: TIMenuItemIntf;
begin
  inherited Create;
  NewMenuItem := nil;
  if ToolServices <> nil then begin
    MainMenu := ToolServices.GetMainMenu;
    if MainMenu <> nil then { main menu }
      try
        MenuItem := MainMenu.FindMenuItem('Borland_FormExpertMenu');
        if MenuItem <> nil then
          try
            MainItem := MenuItem.GetParent;
            if MainItem <> nil then
              try
                NewMenuItem := MainItem.InsertItem(MenuItem.GetIndex+1,
                  '&Dr.Bob''s QueryBob Wizard...','BAddInWizard1','',
                  ShortCut(Ord('D'),[ssShift,ssAlt,ssCtrl]),0,0,
                  [mfEnabled, mfVisible], OnClick)
              finally
                MainItem.DestroyMenuItem
              end
          finally
            MenuItem.DestroyMenuItem
          end
      finally
        MainMenu.Free
      end
  end
end {Create};
```

➤ *Listing 6: Create AddIn Wizard*

```
procedure TBAddInWizard.Execute;
begin
  with TFormWizard.Create(Application) do
    try
      if ShowModal = mrOK then Finish
    finally
      Free
    end
  end {Execute};
```

➤ *Listing 7: Execute AddIn Wizard*



➤ *Figure 4*

```
type
  TQueryTableProducerComponentEditor = class(TComponentEditor)
  public
    function GetVerbCount: Integer; override;
    function GetVerb(Index: Integer): String; override;
    procedure ExecuteVerb(Index: Integer); override;
  end;
```

➤ *Listing 8:*

```
function TQueryTableProducerComponentEditor.GetVerbCount: Integer;
begin
  Result := 1
end {GetVerbCount};
function TQueryTableProducerComponentEditor.GetVerb(Index: Integer): String;
begin
  Result := 'Query-2-HTML CGI-Form Wizard...'
end {GetVerb};
procedure TQueryTableProducerComponentEditor.ExecuteVerb(Index: Integer);
begin
  with TFormWizard.Create(Application) do
  try
    with (Component AS TQueryTableProducer) do begin
      if Assigned(Query) then begin
        ComboBoxAliases.Text := (Query AS TQuery).DatabaseName;
        MemoSQL.Lines.Assign((Query AS TQuery).SQL);
        if Assigned((Query AS TQuery).Params) then
          TheQuery.Params.Assign((Query AS TQuery).Params)
      end
    end;
    if ShowModal = mrOK then
      Finish
  finally
    Free
  end
end {Edit};
```

➤ *Listing 9: ExecuteVerb Component Editor*

```
type
  TBQueryTableProducer = class(TQueryTableProducer)
  private
    FQueryHTML: ShortString;
  published
    property QueryHTML: ShortString read FQueryHTML write FQueryHTML;
  end {TBQueryTableProducer};
```

➤ *Listing 10:*

longer have access to the Action Editor, for example. This might not be a very big problem (just uninstall the package that includes the `TQueryTableProducerComponentEditor` and we then get the original component editor back), but it sure is not the most convenient way to work.

The problem is actually caused by the fact that we have no source code for the 'original' `TQueryTableProducer`'s component editor, so we cannot inherit from it and simply add our new wizard to the list of verbs. The original component editor is probably part of a design-time package and I can understand why Borland isn't shipping source code for the design-time packages.

Property Editor?

A wizard doesn't give us true integration, and a component editor costs us too much 'original' behaviour. So, what's left? Property editors, of course (see Issue 6 for more information). Although this isn't an ideal solution either, as we will see, it gives us the best of both worlds: true integration without too many costs.

We can write a property editor for a specific property of the `TQueryTableProducer` component. This sounds ideal, but which property should we use? The `Query` property, while having the information we need, already has its own property editor and, like component editors, it's not possible to extend the original functionality since we don't have the source code for the Delphi IDE design-time packages. As a matter of fact, I could not find a single property of `TQueryTableProducer` that I could 'reuse' for our purpose, so there was only one option left: add a property of my own to `TQueryTableProducer`. Unfortunately, this also means I need to create a new class, `TBQueryTableProducer`, derived from `TQueryTableProducer`, with the extra property: `QueryHTML` (Listing 10).

For this new property `QueryHTML` of type `ShortString` I can define a special property editor, that will show an ellipsis inside the Object Inspector, and when we click on that, it will perform just like the component editor and wizard we created before, see Listing 11.

The two methods from `TQueryHTMLProperty` we need to override actually specify that we want a dialog property editor (one that shows an ellipsis, and will pop up a dialog when we click on it): see Listing 12.

`GetComponent` returns the `Index`th component being edited by this property editor. This is used to retrieve the `TBQueryTableProducer` component itself, which holds the `Query` property that we need to inspect. Of course, a property editor can only refer to multiple components when `paMultiSelect` is returned from `GetAttributes`, which isn't true in our case, so we can just use `GetComponent(0)` and see if the `Query` property is assigned a value.

The final step involves registering the new property editor, and unlike the wizard and the component editor we did before, this time we also need to register a new component `TBQueryTableProducer`, and we need to specify that the new property editor should only be activated for the `QueryHTML` property of a `TBQueryTableProducer` component: see Listing 13.

The end-result works just like the component editor, but at a lower cost: the only side-effect this time is the new property `QueryHTML` in a new component `TBQueryTableProducer` (which is exactly the same as the base class `TQueryTableProducer`, of course, apart from the new property we have added). Not a perfect solution, of course, but at least it works without actually removing some other needed functionality, such as the Actions Editor in case of the component editor.

Package!

So far, we've created four units: One to hold the wizard code, one for the wizard wrapper, one for the component editor wrapper, and one for the property editor wrapper. Time to wrap things up in one package (pun intended). Packages (see Issue 23) are a convenient way to put all these Delphi IDE extensions together. Of course, you're free to remove the ones you don't want (such as the component editor), but after you've recompiled the package, you only need to copy it to the Delphi 3 BIN directory and install it (use the Component | Install Packages menu) to have the Query-2-HTML CGI-Form Wizard available at design-time whenever you need it: see Listing 14.

The only thing left is showing how to actually use the result together with a Delphi 3 Web Module project: the reason we started this utility in the first place.

Usage

Start a new Web Module project and drop a TBQueryTableProducer from the internet tab on the Web Module. This control will get connected to a parameterised query and generate the dynamic HTML pages from the query result. Right next to the TBQueryTableProducer 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."Common_Name" =
   :"Common_Name")
```

I always give the parameter the same name as the field it connects to, so it's easier to remember (and also easier to convert).

The above parameterised query will return all fields for every record in BIOLIFE where the field Common_Name is equal to a runtime specified common name, as passed in parameter Common_Name. Next, we need to specify the type of this parameter by clicking on the Params property in the Object Inspector. The type is String, and there is no default value. We can test the SQL syntax by double-clicking on the Active property of the TQuery component. If it gets set to True we've built a valid SQL query.

Now that we've entered the Query, it's time to connect it to the TBQueryTableProducer. Just click on the TBQueryTableProducer component on the Web Module, go to the Object Inspector, and set the Query property to Query1. Now the TBQueryTableProducer will use the resulting database of the TQuery to generate the dynamic HTML pages. And that's not all. The TBQueryTableProducer is not only able to get the TQuery result, it is also able to set the Query input parameters (Common_Name in our case).

This is where our Query-2-HTML CGI-Form converter comes in. Based on the information specified in the TQuery component, we can now generate an HTML CGI-Form that contains an INPUT field with name Common_Name (ie the name of the parameter inside the Query). Whenever the data from the form is sent to the web application, a

```
type
  TQueryHTMLProperty = class(TStringProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    procedure Edit; override;
  end;
```

➤ *Listing 11:*

```
function TQueryHTMLProperty.GetAttributes: TPropertyAttributes;
begin
  Result := [paDialog]
end {GetAttributes};
procedure TQueryHTMLProperty.Edit;
begin
  with TFormWizard.Create(Application) do
    try
      with (GetComponent(0) AS TQueryTableProducer) do begin
        if Assigned(Query) then begin
          ComboBoxAliases.Text := (Query AS TQuery).DatabaseName;
          MemoSQL.Lines.Assign((Query AS TQuery).SQL);
          if Assigned((Query AS TQuery).Params) then
            TheQuery.Params.Assign((Query AS TQuery).Params)
        end
      end;
      if ShowModal = mrOK then begin
        SetValue(EditFileName.Text);
        Finish
      end
    finally
      Free
    end;
  Modified
end {Edit};
```

➤ *Listing 12: Property Editor*

```
procedure Register;
begin
  RegisterComponents('internet',[TBQueryTableProducer]);
  RegisterPropertyEditor(TypeInfo(ShortString), TBQueryTableProducer,
    'QueryHTML', TQueryHTMLProperty)
end {Register};
```

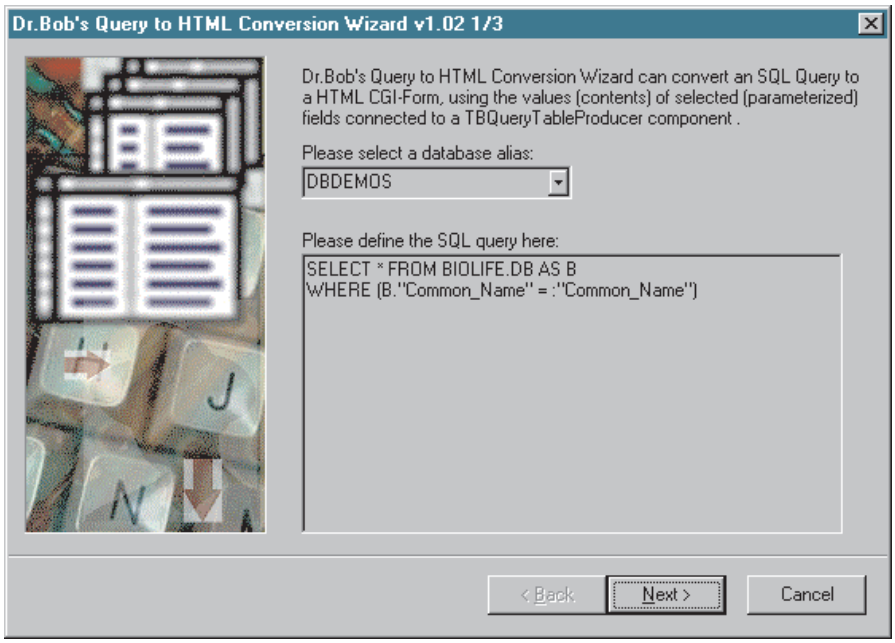
➤ *Listing 13:*

```
package BobWeb30;
{$DEBUGINFO OFF}
{$LOCALSYMBOLS OFF}
{$LONGSTRINGS ON}
{$OPTIMIZATION ON}
{$OVERFLOWCHECKS OFF}
{$RANGECHECKS OFF}
{$STACKFRAMES OFF}
{$REFERENCEINFO OFF}
{$TYPEDADDRESS OFF}
{$VARSTRINGCHECKS ON}
{$WRITEABLECONST ON}
{$IMAGEBASE $00400000}
{$DESCRIPTION 'Dr.Bob''s Web Module
  Enhancements'}
{$DESIGNONLY}
{$IMPLICITBUILD ON}
requires
  vcl30,
  VCLX30,
  VCLDB30,
  INETDB30,
  INET30;
contains
  QueryBob, { IDE Expert/Wizard }
  DrBobCED, { Component Editor }
  DrBobWeb, { Property Editor }
  DrBobWiz;
end.
```

➤ *Listing 14: BobWeb30 Package*

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := BQueryTableProducer1.Content;
end;
```

➤ *Listing 15:*



➤ *Figure 5*

match is made between the `INPUT` field names and the `Query` parameter names. If a match is found, the value of the `INPUT` field is substituted for the `Query` parameter. In our case, we just generate a simple CGI form that contains an `INPUT` field with the name `Common_Name` to hold the Common Names of the fish from the `BIOLIFE` table.

If we click in the Object Inspector on the ellipsis for the `QueryHTML` property, we get our wizard, only this time it can extract all required information from the `TQuery` itself, including the parameterised query and all its parameters (Figure 5).

All we need to do now is create a default `WebActionItem`, and make sure the `OnAction` event contains the code in Listing 15 to redirect output from the `TBQueryTable Producer` to the final output of the web application. As long as we remember to keep the `Active` property of the Query set to `True`, we don't need to write any more code.

Home Improvements

Now that we have a tool to generate a static HTML form for query support, it occurs to me that it might be useful to be able to generate this form dynamically as well. Since tables can change, this may solve the potential problem of looking for a `Common_Name`, which isn't in the database anymore.

Generating the HTML form dynamically shouldn't be hard to do, given the 'building blocks' we now have. We must add a special action to our Web Module, which then generates the HTML form, instead of pointing to a static copy on the net. The only problem I can see would be how to determine which fields to include in the generated form. But maybe that can be specified in the 'mother of all queries' form, in which one could specify the table and fields.

I'm still not happy that we ended up with a property editor instead of a component editor for `TQuery-TableProducer`. I'm hoping that I'll get a clue (email bob@bolesian.nl) to help me solve it...

Next Time, Dr.Bob Says...

There are a number of web pages on my bookmark list that I plan to visit regularly. But sometimes I just don't have the time and so may miss an important announcement. At other times I do visit a site but nothing has changed. Ever had this experience? Let me rephrase that last question: What do the internet, Intelligent Agents, Delphi and Dr.Bob have in common? Learn the answer next time when a new internet 'searching' tool is born called... *RobotBob*.

Bob Swart (aka Dr.Bob, www.drbob42.com) is a professional knowledge engineer technical consultant using Delphi, C++Builder and JBuilder for Bolesian, a freelance technical author and co-author of the web-based *Delphi Internet Solutions* knowledgebase.