Under Construction: Internet Database Applications

by Bob Swart

In this article we'll create a real database application on the internet, using simple CGI techniques, *without* the Delphi 3 Web Modules. Next time, we'll enhance this solution even further by upgrading the CGI application to an ISAPI web server extension DLL.

Whilst the Delphi 3 Client/Server Edition Web Modules offer a significant enhancement over the 'traditional' development of CGI and ISAPI/NSAPI internet applications, not everyone is able to purchase the Delphi 3 Client/Server Suite. Many developers are using the Professional Edition.

So, in response to readers' requests, this month we will develop a database (actually, a single table for now) and create CGI applications to browse, edit, insert, delete and query this database, including multi-user awareness. As the main example for this month, I decided for once not to use the BIOLIFE example table *[Hurrah! Ed]*, but something more interesting, and come up with a final application that might even be useful for a number of people.

Long time subscribers (or people who purchased *The Delphi Magazine Collection '96* CD-ROM) may remember Issue #11, where we created an AddIn wizard for error reporting and error handling, based on a single table. This wizard, called BERT (for Bolesian Error Report Tool), has had some minor enhancements since then, and is now ready to undergo a final upgrade to become internet-aware. Using a web browser we can now report errors and query the progress of error fixes too.

Database

The single REPORT.DB table to store the reports consists of an integer key field called `Report`, a number of string fields (mainly items picked from a list), two date fields and two memo fields (for detailed comments). The Object Pascal source code to generate the table is on the disk.

Pick-Lists

Now that we have the table in which to store our reports, we need to define the lists of items for a number of the string fields. At least for `ReportStatus`, `ReportSystem`, `ReportProblem`, `ReportPriority` and `FixStatus` we can specify a set of pre-defined answers. If we have a limited number of testers and/or bug-fixers then the same can be said for `ReportedBy` and `FixedBy`.

Using a REPORT.INI file I can define the entries in the pick-lists (Listing 1). Also, given the names of the fields in the table, I would also like to specify some more suitable display names to be shown to the user. For each field I defined a section in REPORT.INI, with entries for `Name` and `Items` (if it contains a list of pre-defined items). Given the fact that I want to dynamically produce a set of HTML forms, it would help if I could specify the form action and also a title and bitmap to place on top of each page. This is done in the general `[report.db]` section of the INI file.

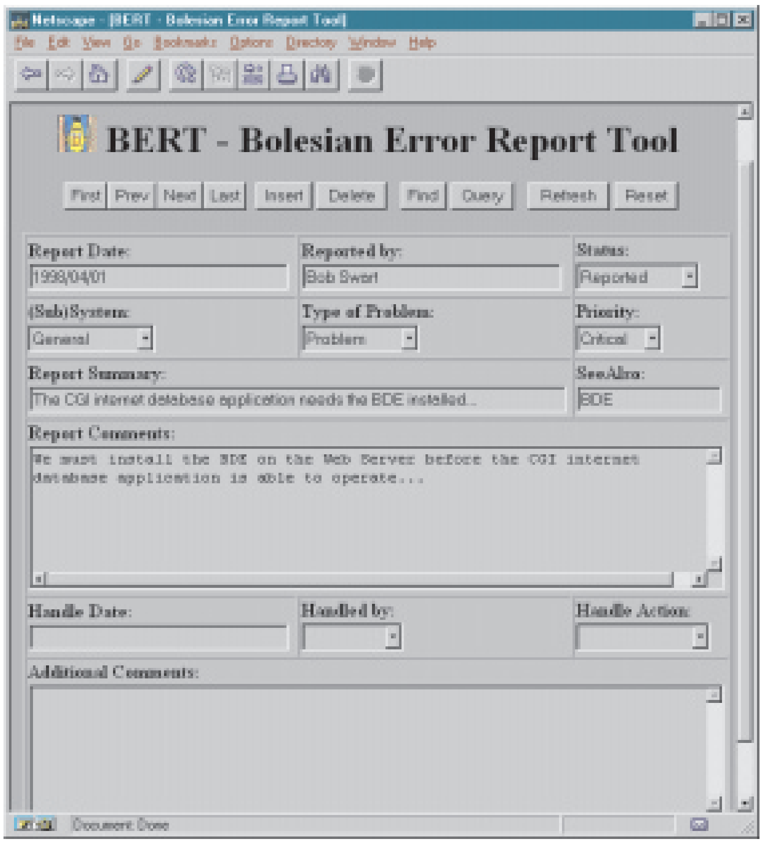
Target

Ideally, the application should look like Figure 1 when browsing records: ready to go to the first, previous, next or last record, ready to insert or delete a record, to find a record or perform some query, refresh a record (if someone else has changed its value for example) or to reset the input on the form if we typed new data.

The first step in making this work is to write a little program to generate the HTML form that makes up Figure 1. Note that, apart from 'action' buttons, the only control types that I use are text (plain editbox), drop-down comboboxes

➤ *Listing 1*

```
[report.db]
Name=BERT - Bolesian Error Report Tool
Bitmap=http://www.bolesian.nl/groups/
  delphi/bolesian/gif/bert-ico.gif
Action=http://www.bolesian.nl/cgi_bin/
  bert.exe
[ReportDate]
Name=Report Date:
[ReportedBy]
Name=Reported by:
[ReportStatus]
Name=Status:
Items=5
Item1=Reported
Item2=Open (active)
Item3=Handled
Item4=Verified
Item5=Closed
[ReportSystem]
Name=(Sub)System:
Items=5
Item1=General
Item2=Database
Item3=Engine
Item4=User Interface
Item5=Reports
[ReportProblem]
Name=Type of Problem:
Items=5
Item1=Error (Crash)
Item2=Problem
Item3=New Wish
Item4=Remark
Item5=Question
[ReportPriority]
Name=Priority:
Items=5
Item1=Critical
Item2=High
Item3=Medium
Item4=Low
Item5=None
[ReportSummary]
Name=Report Summary:
[ReportSeeAlso]
Name=SeeAlso:
[ReportComments]
Name=Report Comments:
[FixDate]
Name=Handle Date:
[FixedBy]
Name=Handled by:
Items=2
Item1=
Item2=Bob Swart
[FixStatus]
Name=Handle Action:
Items=4
Item1=
Item2=Fixed
Item3=WAD (ignored)
Item4=New Spec.
[FixComments]
Name=Additional Comments:
```


Netscape - [BERT - Bolesian Error Report Tool]
File Edit View Go Bookmarks Options Directory Window Help
First Prev Next Last Insert Delete Find Query Refresh Reset
Document: Done

BERT - Bolesian Error Report Tool

Report Date: 1998/04/01
Reported by: Bob Swart
Status: Reported
(Sub)System: General
Type of Problem: Problem
Priority: Critical
Report Summary: The CGI internet database application needs the BDE installed...
SeeAlso: BDE
Report Comments: We must install the BDE on the Web Server before the CGI internet database application is able to operate...
Handle Date:
Handled by:
Handle Action:
Additional Comments:

➤ *Figure 1*

and a text field (for a multi-line memo).

Yet Another HTML CGI Wizard

We probably all know by now that an HTML CGI Form is embedded within `<FORM ACTION=... METHOD=...>` and `</FORM>` tags. And inside these tags, we can define the different input fields. A single `<INPUT TYPE=text>` for plain edit boxes, a set of `<TEXTAREA ROWS=... COLS=...>` and `</TEXTAREA>` for a memo field, and a set of `<SELECT>` and `</SELECT>` tags, with `<OPTION>` tags inside, for each item in a combobox. Given the REPORT.DB table, we can analyse the fields inside this table, and determine whether or not a field is of type ftMemo (so we use a memo field instead of a simple field):

```
with DataSet do
  for i:=0 to FieldCount-1 do
    if Fields[i].DataType =
      ftMemo then { memo field }
```

If we're not dealing with a memo field, then we need to take a look at the INI file for this table and find out if the value of the `Items` entry in the section for this particular field-name has a value bigger than zero. This can be done as follows:

```
with DataSet, IniFile do
  if ReadInteger(Fields[i].FieldName,
    'Items', 0) = 0 then
    { no items = editbox }
```

And in case we stumble upon a field that has a certain number of pre-defined answer items associated with it, we just have to read each item from the INI file (in sections `Item1` to `ItemXX` where `XX` is the number of items defined). This can be seen in Listing 2 in the procedure `DataSetTable`.

Before we think we're done for now, however, I want to make sure that each field is positioned in a 'nice' and 'user friendly' way somewhere on the HTML form. To implement this, I use a `<TABLE>` with

➤ *Listing 2*

```
procedure DataSetTable(DataSet: TDataSet);
const
  Int: Array[1..9] of Char = '123456789';
var
  i,j,col,items: Integer;
  option: ShortString;
begin
  with DataSet do begin
    writeln('<TABLE BORDER><TR>');
    col := 0;
    with TIniFile.Create(IniFile) do
      { IniFile = TableName with '.INI' extension }
      try
        for i:=1 to FieldCount-1 do begin
          { ignore first field }
          if Fields[i].DataType = ftMemo then begin
            { memo = 3 columns }
            writeln('</TR><TR><TD COLSPAN=3>');
            col := 3
          end else if Fields[i].Size > 99 then begin
            { wide field = 2 columns }
            Inc(col,2);
            if col > 3 then begin
              writeln('</TR><TR>');
              col := 2
            end;
            write('<TD COLSPAN=2>')
          end else begin
            { small field = 1 column }
            Inc(col);
            if col > 3 then begin
              writeln('</TR><TR>');
              col := 1
            end;
            write('<TD>')
          end;
          write('<B>',ReadString(Fields[i].FieldName,
            'Name',Fields[i].FieldName), '</B><BR>');
          items :=
            ReadInteger(Fields[i].FieldName,'Items',0);
          if items = 0 then begin
            { memo or text field }
            if Fields[i].DataType = ftMemo then begin
              writeln('<TEXTAREA NAME="',
                Fields[i].FieldName, '" ROWS=6 COLS=72>');
              writeln('</TEXTAREA>')
            end else
              writeln('<INPUT TYPE=text NAME="',
                Fields[i].FieldName,
                '" SIZE=',Fields[i].Size)
          end else begin
            { combobox: items > 0 }
            writeln('<SELECT NAME="',
              Fields[i].FieldName,'">');
            for j:=1 to items do begin
              option := ReadString(Fields[i].FieldName,
                'Item'+Int[j],Int[j]);
              writeln('<OPTION VALUE="',option,'">',
                option,' ')
            end;
            writeln('</SELECT>')
          end;
          writeln('</TD>')
        end;
        writeln('</TR>')
      finally
        writeln('</TABLE>');
        Free
      end
  end
end;
```

```
<FORM ACTION="http://www.bolesian.nl/cgi_bin/bert.exe" METHOD=POST>
<INPUT TYPE=SUBMIT NAME=Action VALUE=First>
<INPUT TYPE=SUBMIT NAME=Action VALUE=Prev>
<INPUT TYPE=SUBMIT NAME=Action VALUE=Next>
<INPUT TYPE=SUBMIT NAME=Action VALUE=Last>
<INPUT TYPE=SUBMIT NAME=Action VALUE=Insert>
<INPUT TYPE=SUBMIT NAME=Action VALUE=Delete>
<INPUT TYPE=SUBMIT NAME=Action VALUE=Find>
<INPUT TYPE=SUBMIT NAME=Action VALUE=Query>
<INPUT TYPE=SUBMIT NAME=Action VALUE=Refresh>
<INPUT TYPE=RESET VALUE=Reset>
...
</FORM>
```

➤ *Listing 3*

a border and three columns. By default, each field uses one line in one column. Exceptions are big fields (ie string fields over 99 characters) which take up two columns, and memo fields, which take up the entire row of three columns.

The code in Listing 2 can be used to create an empty HTML CGI Form for any dataset (TTable or TQuery). If we have an additional INI file, we can even produce comboboxes with predefined answer values, and names to use in the form instead of the actual field names.

Note that each CGI input field gets the exact name of the field in the table, so we're fully prepared already to write some 'connecting' code between the CGI input fields and a web server application.

CGI Executable

Now that we've written the code to deal with the client side, it's time to focus on the server side: the CGI application itself that gets information from the form to perform the actions. These actions are represented by the line of 10 buttons on top of the form, as in Figure 1.

Normally, there are two kinds of buttons in a form: RESET and SUBMIT (see Listing 3). Fortunately, we can have more than one SUBMIT button, as long as we give each of them a NAME (such as Action) and a unique VALUE. In that case, the VALUE is passed as the CGI value for the NAME (in this case Action) CGI variable.

CGI Input Variables

Most web servers support the standard CGI protocol, which means that in order to obtain the data passed from the client to the server application, we must first check the environment variable REQUEST_METHOD, to see if the POST or GET protocol was used. In case of a GET, we must parse the URL itself, while a more flexible POST means we need to read a certain number of characters from the standard input. The exact number of characters is stored in the value of the CONTENT_LENGTH environment variable. The data that we receive this way is encoded in a special way: spaces have been replaced by a plus characters (+), and special characters have been encoded using a percent sign (%) followed by two hex digits. Once we've decoded the data, we can extract the individual fields and values which are stored as field=value pairs, separated by an ampersand.

I wrote the unit DrBobCGI to handle these technical details, providing one function called Value that we can use to obtain the value of a CGI variable (passed as an argument). Quite a useful unit, if I may say so, as it shields us from the CGI protocol details, offering the potential of upgrading to another protocol (say, ISAPI) without having to change the user/caller of the Value API itself.

Using the DrBobCGI unit, we can get the value of the Action CGI field, and determine the action (First, Prev, Next, Last, Insert, Delete, Find, Query or Refresh). If for some reason we don't get a value for the Action field, we give it a default value of First and show the first record of the REPORT.DB table.

```
{ determine Action }
Action := Value('Action');
if Action = '' then
  Action := 'First';
```

Hidden Information

Assuming we want to fix the DatabaseName and TableName information for our CGI application (which means we need to write a unique CGI application for every table in our database), we do need to keep track of the current record pointer. And while the current record number may sound good enough, this won't work if someone deletes or inserts a record. Instead, we must determine the key values of the table, and store the key values as hidden fields inside the form, so we can determine and locate the current record before we attempt to execute the action we need to perform.

For the REPORT.DB table, the key field only consists of the first (integer) field called Report, which we can store as a hidden field as follows:

```
<INPUT TYPE=HIDDEN
  NAME="Report" VALUE="1">
```

Using this hidden field, we can locate the current record (which we must do before we perform the action) as follows:

```
{ locate current record }
Report := Value('Report');
if Report > 0 then
  Table.FindKey([Report]);
```

Now it's time for action...

Browsing

For browsing actions (First, Prev, Next and Last) the operation is easy: just perform the operation on the table followed by a call to the DataSetTable routine (but this time a version that also lists the value of the fields in the current record). See Listing 5.

Locate

For the Find and Query buttons, we may want to do something special, like showing another form to enter search strings.

I'll skip these for now and we'll get back to them next time and then design some HTML forms to handle these actions:

```
else if (Action = 'Find')
  or (Action = 'Query')
  then begin
    // special
end
```

```
unit DrBobCGI;
{$I-}
interface
function Value(const Field: ShortString): ShortString;
{ use this API to obtain the CGI field value for "Field" }
implementation
uses
  SysUtils, Windows;
type
  TRequestMethod = (Unknown,Get,Post);
var
  RequestMethod: TRequestMethod = Unknown;
  ContentLength: Integer = 0;
  Data: AnsiString = '';
function Value(const Field: ShortString): ShortString;
  var
    i: Integer;
    len: Byte absolute Result;
  begin
    Len := 0;
    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] <> '&' do begin
        Inc(Len);
        Result[Len] := Data[i];
        Inc(i)
      end
    end
  end {Value};
var
  P: PChar;
  i: Integer;
  Str: ShortString;
initialization
  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;
    Inc(P, StrLen(P)+1)
  end;
  if RequestMethod = Post then begin
    SetLength(Data,ContentLength+2);
    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] := ' ';
    if Data[i] = '%' then begin
      { special hex code }
      Str := '$00';
      Str[2] := Data[i+1];
      Str[3] := Data[i+2];
      Delete(Data,i+1,2);
      Data[i] := Chr(StrToInt(Str))
    end
  end;
  if i > 0 then
    Data[i+1] := '&'
  else
    Data := '&'
finalization
  Data := ''
end.
```

➤ *Listing 4*

Refresh

The `Refresh` button connects to the table again to obtain the value of the current record. Note that another user may have changed the current record while we were waiting, and in that case `Refresh` will always give you the latest information from the table.

Insert and Delete

Clicking on `Delete` can be translated into a simple `Delete` operation on the table again. Clicking on `Insert` means we must get a similar HTML form, but this time with all the fields empty so the user can enter new values and then click on a `Post` or `Cancel` button. This second form can look like Figure 2.

This form only has two buttons: one which has `Post` as value for the Action, and one with `Cancel` as value for the action. In case we fill in each field and click on the `Post` button, we need to actually insert a new record in the database, using a new (highest) report number, giving each field a possible value, and finally posting this new record (Listing 6).

➤ *Listing 5*

```
{ perform action }
if Action = 'First' then
  First
else if Action = 'Next' then
  Next
else if Action = 'Prev' then
  Prior
else if Action = 'Last' then
  Last
```

There is one thing we have to keep in mind with the Listing 5 code: the call to `Post` can fail, especially with `Date` fields that are incorrectly formatted (for example when you enter 13/4/1998 while the `ShortDate` format is defined as `DD/MM/YYYY`). This is something we could respond to at this point in code, or perhaps by adding some JavaScript code to the web page to check the input (thereby doing data entry validation on the client instead of the server).

DataSetTable

After we've performed the action, it's time to call the enhanced `DataSetTable` function again, where the second (`Boolean`) argument specifies whether or not we want to show empty fields only (in case we are inserting a new record into the table):

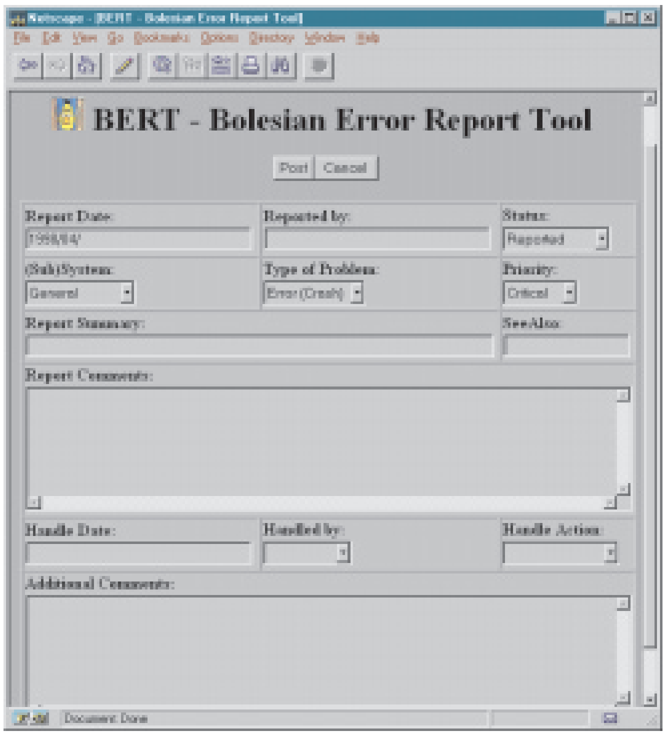
```
{ generate HTML CGI-Form
  with fields }
DataSetTable(Table,
  Action = 'Insert');
Close;
```

After the HTML page has been generated, we can close the table again, and terminate the CGI application at the web server, which will then send the generated HTML to the user and his web browser at the client side.

Edit/Modify

There's one more thing we should take into account: the ability to change the values of fields of the current record while we're browsing the database. This should be possible at all times.

The easiest solution is to add an `Update` button to the HTML CGI-Form, which does the same as the `Post` button, but doesn't start looking for a new unique key value. Rather, it uses the current key value, since we only want to update the current record. When updating an existing record this


Netscape - [BERT - Bolesian Error Report Tool]
File Edit View Go Bookmarks Options Directory Window Help
BERT - Bolesian Error Report Tool
Post
Cancel
Document: Done

Report Date: 1998/04/	Reported by:	Status: Reported
(Sub)System: General	Type of Problem: Error (Crash)	Priority: Critical
Report Summary:		SeeAlso:
Report Comments:		
Handle Date:	Handled by:	Handle Action:
Additional Comments:		

➤ *Figure 2*

way, we should also need to check if the record didn't change (behind the scenes) during the time we were editing the record on our client browser. In order to ensure this, we must somehow make sure the HTML form not only contains the new updated values, but also the original values of each field in the record. The latter information should be hidden, but present nevertheless. This means that we have to extend the code that only writes the values of the key fields (as hidden fields) to perform this for every field in the record, as shown in Listing 7.

Note that we prefix the names of the fields with an underscore, to distinguish them from actual field names.

The `Update` action now first needs to check the value of the current record against the 'old' values as found inside the HTML form. If they indeed match, then an update can be done with the new values. If not, then a message should be returned instead, telling the user that the update could not be performed, because someone else changed the record just a little bit earlier.

Note that this is actually an enhancement compared to the current standard way of having multiple users accessing the same tables (where they can change the same record, and the last update will overwrite an earlier one).

If we don't want to use the `Update` button, we should be prepared to expect an update with every action the user takes (ie while browsing, finding, etc), by first comparing the 'old' field values to the 'new' ones and, if they differ, prepare to update the record in the table (otherwise just perform the original action). This is actually a situation that more closely corresponds to the Delphi `TDBNavigator` control and usage, and can be implemented as shown in Listing 8.

This introduces another problem. Comparing the contents of a memo field in a table with a `<TEXTAREA>` proved to be difficult, introduced by the number of carriage return and line feed characters (CR and LF). So, I decided to modify the `Value` function inside the `DrBobCGI` unit to skip all CR and LF characters and replace them by a single blank character instead (see the updated unit `DrBobCGI` on the disk for details). Fortunately, this solved the problem.

The final code for the CGI internet database application is on this month's disk.

Performance

Running the CGI database internet application from within a web browser shows the results as seen in the two figures.

However, there is one significant problem: performance. And that's not at all strange, considering that, for each and every request, the CGI application has to be loaded, the entire BDE has to be loaded as well, the table has to be opened, the current record must be found, followed by an optional update of the current record (if the data sent by the user has changed from the original version), the action is performed (depending on which of the buttons the user clicked on), the HTML form is dynamically generated again with all the information, and finally the CGI applications and the BDE have to be shut down. Phew!

➤ *Listing 6*

```
else if Action = 'Delete' then
  Delete
else if Action = 'Insert' then
  { skip - DataSetTable will show new empty fields }
else if Action = 'Post' then begin
  { insert record }
  First;
  Report := 0;
  while not Eof do begin
    { find highest Key value in use }
    if Fields[0].AsInteger > Report then
      Report := Fields[0].AsInteger;
    Next
  end;
  Inc(Report);
  Insert;
  Fields[0].AsInteger := Report;
  for i:=1 to FieldCount-1 do
    Fields[i].AsString := Value(Fields[i].FieldName);
  Post
end else if Action = 'Cancel' then
  { Cancel - ignore }
else
  { Refresh - ignore };
```

➤ *Listing 7*

```
for i:=0 to FieldCount-1 do
  writeln('<<INPUT TYPE=HIDDEN NAME="_',Fields[i].FieldName,
    '" VALUE="',Fields[i].AsString,'">>');
```

```
{ update record if data has changed }
if (Value('_'+Fields[0].FieldName) <> '') and
  { old data is stored } (ValueAsInteger(Fields[0].FieldName) <> -1)
  then begin
  NoChange := True; { assume no change }
  for i:=0 to FieldCount-1 do
    NoChange := NoChange AND
      (Value('_'+Fields[i].FieldName) = Value(Fields[i].FieldName));
  if not NoChange then begin
    { update record. check if data in table is still the same }
    NoChange := True;
    for i:=0 to FieldCount-1 do
      NoChange := NoChange AND
        (Value('_'+Fields[i].FieldName) = Fields[i].AsString);
    if not NoChange then begin
      { table changed!! }
      writeln('<B>Error: value of record changed before '+
              'your update was made!</B>');
      Action := 'Refresh' { force refresh }
    end else begin
      { go ahead! }
      writeln('<FONT SIZE=2>Note: ');
      Edit; { set Table in Edit-mode }
      for i:=0 to FieldCount-1 do begin
        if (Value('_'+Fields[i].FieldName) <> Value(Fields[i].FieldName))
          then begin
         {$IFDEF DEBUG}
          write(i,' [',Value('_'+Fields[i].FieldName),
              ']-{',Value(Fields[i].FieldName),'} ');
         {$ENDIF}
          Fields[i].AsString := Value(Fields[i].FieldName) { new }
        end
      end;
      Post { Post data in Table };
      writeln(' previous record updated in table</FONT><P>')
    end
  end
end;
```

➤ *Listing 8*

The load and unload times especially weigh heavily on the response times of this simple internet database CGI example.

Next Time

Next month we'll design some nice HTML forms to find a given record in a database table, or to perform a dynamic SQL query and display the results too.

Also, we'll discover how we can improve the performance of the simple CGI database application by turning it into an ISAPI application. Technically, this should only mean a few changes: the application itself must become a DLL, must conform to the ISAPI API, and should probably use a slightly different method of obtaining the values of the CGI variables.

Other than that, we'll see how we must change the way we work with the Borland Database Engine in order to prevent clashes between multiple simultaneous users. All this and more too, so stay tuned...

Bob Swart (aka Dr.Bob, www.drbob42.com) is a knowledge engineer technical consultant using Delphi, C++Builder and JBuilder for Bolesian. Bob also likes to watch videos of Star Trek Deep Space Nine and Voyager with his 4 year old son Erik Mark Pascal and 1.5 year old daughter Natasha Louise Delphine.