DDG Bug-Reporting Tool: Using WebBroker

CHAPTER

36

IN THIS CHAPTER

The last chapter, "DDG Bug Reporting Tool: Desktop Application Development," demonstrated various techniques for designing desktop database applications. One consideration we discussed was how to develop an application that you plan to deploy to the World Wide Web. In this chapter, we are going to deploy the last chapter's application, a simple bug-reporting tool, to the World Wide Web as an ISAPI server. As stated in the previous chapter, this effort should require minimal modifications to the code already written. We will use the techniques covered in Chapter 31, "Internet-Enabling Your Applications with WebBroker." Therefore, we will not go into any detail here on topics covered in that chapter. If you feel you need to review Chapter 31, you might do so before reading on.

The Page Layout

The layout (flow) of this Web-based bug-reporting tool is illustrated in Figure 36.1.

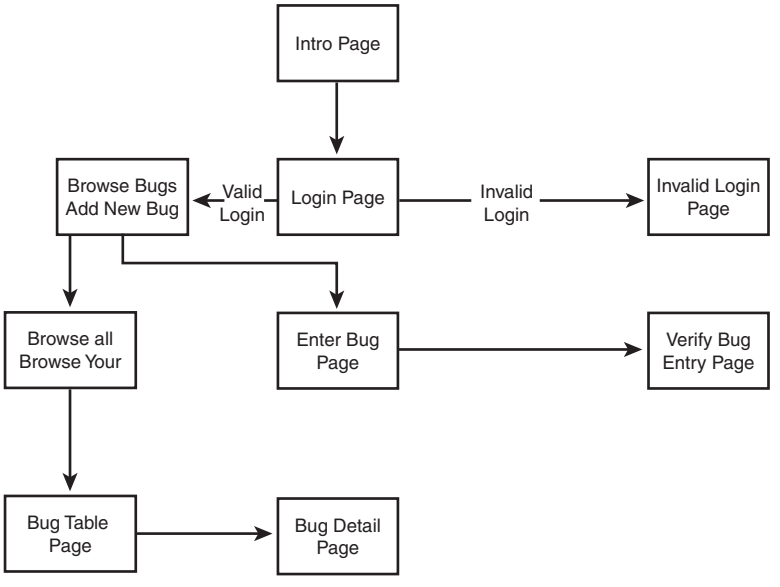

FIGURE 36.1
The flow for the Web-based bug-reporting tool.

You can see from the page layout that this application is really a subset of the functionality presented in Chapter 35. As an exercise, feel free to expand on the techniques demonstrated in this chapter to provide the full functionality presented in the previous chapter.

The following sections explain the code used to develop the pages. You will notice in this example that all pages are created at runtime—that is, no predefined HTML documents are

loaded. There weren't any compelling reasons why we chose this method instead of writing some HTML documents that are loaded by the WebBroker components. You can certainly use the latter approach for your applications.

Changes to the Data Module

Our intent here is to use much of the functionality and components we used in designing `TDDGBugsDataModule` from the last chapter. We mainly want to add functionality to that data module and minimize any changes that could potentially break its use in the original non-Web-based application. We accomplish this by avoiding making changes to already existing methods. We also recompile and test the original application to further verify that the previous application is left intact.

Note that we did not have to create a separate Web module; rather, we just added the `TWebDispatcher` component to the existing `TDataModule`. This allows us to use `TDataModule` as we had already designed it.

For this Web-based version of the bug-reporting tool, we have added four more components to `TDDGBugsDataModule`: `TWebDispatcher`, `TDataSetTableProducer`, `TPageProducer`, and `TSession`. We will use these components throughout the code.

We should also mention the purpose of the `TSession` component. The ISAPI server DLL can potentially be accessed by multiple clients, meaning that multiple people might be trying to hit the database simultaneously through this single DLL instance. This DLL will operate within a single process space. Therefore, each client that attempts to hit the server requires a separate, dedicated Web module. These separate Web modules are created at runtime and are handled in their own unique thread. This also necessitates each database connection getting its own `TSession` component in order to prevent database connections from conflicting with each other when multiple clients hit the database. By setting the `TSession.AutoSessionName` property of the `TSession` component to `True`, we ensure that each `TSession` instance is also given its own unique name. Actually, it is the thread that requires its own BDE session.

Note that adding a `TSession` component to the Web module or to `TDataModule` is not required when writing a WinCGI or CGI server application, because these are compiled to separate applications that operate in their own process spaces.

Setting Up the TDataSetTableProducer Component: dstpBugs

The data module's `TDataSetTableProducer` component, `dstpBugs`, is attached to the `TTable` component, `tblBugs`. Much like configuring a `TDBGrid`, we have modified the

`dstpBugs.Columns` property to specify titles other than the default (see Figure 36.2). These are the titles that will show up in the Web page table. We have also modified the `dstpBugs.TableAttributes` property to allow for a one-pixel wide border that will give the table a three-dimensional appearance on most Web browsers.

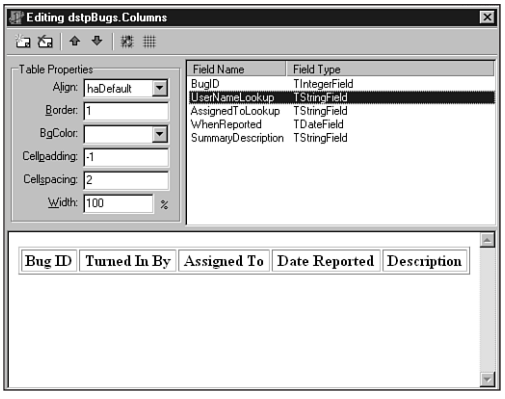

FIGURE 36.2
Editing the `Columns` *property for* `dstpBugs`.

Setting Up the TWebDispatcher Component: wbdpBugs

Figure 36.3 shows the Actions editor used to add several `TWebActionItem` instances to `wbdpBugs`. We will get into the details of each of these actions as well as how they present the user with access to the bug application through the Web.

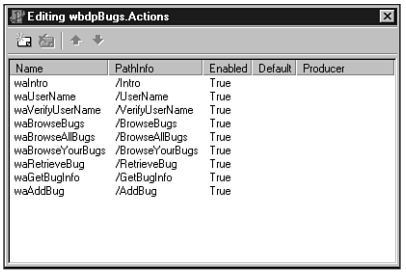

FIGURE 36.3
Editing the `Actions` *property for* `wbdpBugs`.

Setting Up the TPageProducer Component: pprdBugs

If you bring up the `pprdBugs.HTMLDoc` property, you will notice that it is empty. This property is manipulated at runtime programmatically. We will use this same instance of `TPageProducer` in two different situations, as you will see when we discuss the code.

Coding the DDGWebBugs ISAPI Server: Adding TActionItem Instances

All the functionality of the Web bug-reporting tool is provided through the `TWebDispatcher` component's `TActionItem` instances. Table 36.1 shows the purpose of each `TActionItem` instance. We will discuss each of these separately.

TABLE 36.1 The Purpose of the `TActionItem` Instances

`TActionItem`	*Purpose*
`waIntro`	Displays an initial introductory page to the user.
`waUserName`	Prompts the user to enter a username.
`waVerifyUserName`	Invoked from `waUserName.OnAction`. Verifies the username entered by the user.
`waBrowseBugs`	Displays two selections to the user: Browse All Bugs and Browse User's Bugs Only.
`waBrowseAllBugs`	Displays a table containing all the bugs in the database.
`waBrowseYourBugs`	Displays a table containing bugs belonging to the user.
`waRetrieveBug`	Displays detail information on the bugs.
`waGetBugInfo`	Provides the input page to which the user enters new bug information.
`waAddBug`	Adds the new bug to the table and displays a verification screen.

In the following sections, we will show the individual listing for each method added to the `DDBBugsDM.pas` unit instead of showing the entire listing.

Helper Routines

The `AddHeader()` procedure, shown in Listing 36.1, is used to add a standard header to the Web bug pages consisting of the page title and header. Also, the background image to use is specified here. Note that the location of this background image is dependant on the Web server. You will most likely have to modify this statement, depending on your system, to be able to

find the image. `AddFooter()`, shown in Listing 36.2, is used to add the standard footer information, including the copyright statement.

LISTING 36.1 `TDDGBugsDataModule.AddHeader()` Is Used to Add the Standard Header Information

```
procedure AddHeader(AWebPage: TStringList);
// Adds a standard header to each web page.
begin
  with AWebPage do
  begin
    Add('<HTML>');
    Add('<HEAD>');
    Add('<BODY BACKGROUND=''/samples/images/backgrnd.gif''">');
    Add('<TITLE>Delphi 5 Developer''s Guide Bug Demo</Title>');
    Add('<CENTER>');
    Add('<P>');
    Add('<FONT SIZE=6>Delphi 5 Developer''s Guide Bug Demo</font>');
    Add('</CENTER>');
    Add('</HEAD>');
  end;
end;
```

LISTING 36.2 `TDDGBugsDataModule.AddFooter()` Is Used to Add the Standard Footer Information

```
procedure AddFooter(AWebPage: TStringList);
// Adds the standard footer information to each web page.
begin
  with AWebPage do
  begin
    Add('<BR><BR>Copyright (c) 1998, Delphi 5 Developer''s Guide.');
    Add('</BODY>');
    Add('</HTML>');
  end;
end;
```

The Introduction Page

The introduction page is shown in Figure 36.4. It is created by the `waIntro.OnAction` event handler, `wbdpBugswaIntroAction()`, which is shown in Listing 36.3.

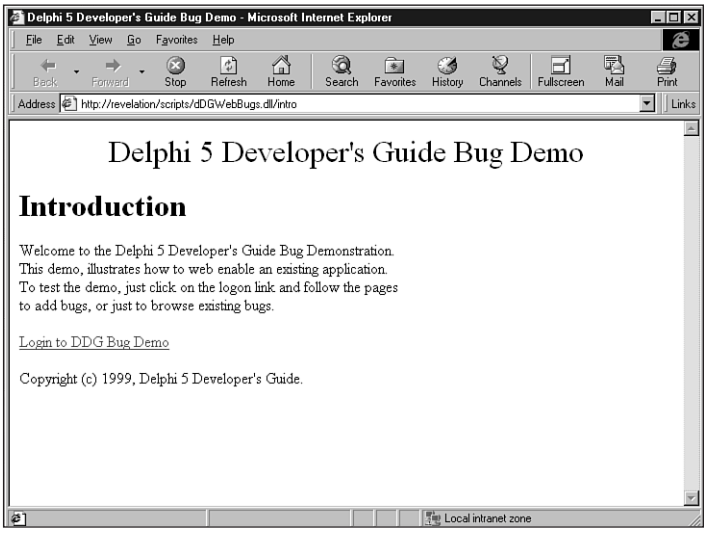

FIGURE 36.4
The Introduction page.

LISTING 36.3 `TDDGBugsDataModule.wbdpBugswaIntroAction()` Displays an Initial Introductory Page

```
procedure TDDGBugsDataModule.wbdpBugswaIntroAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
// Introductory page for the web demo.
var
  WebPage: TStringList;
begin
  WebPage := TStringList.Create;
  try
    AddHeader(WebPage);
    with WebPage do
    begin
      Add('<BODY>');
      Add('<H1>Introduction</H1>');
      Add('<P>Welcome to the Delphi 5 Developer''s Guide Bug Demonstration.');
      Add('<BR>This demo, illustrates how to web enable an existing
➥application.');
      Add('<BR>To test the demo, just click on the logon
➥link and follow the pages');
      Add('<BR>to add bugs, or just to browse existing bugs.');
      Add('</P>');
```

continues

LISTING 36.3 Continued

```
      Add('<A href="../DDGWebBugs.dll/UserName">Login to DDG Bug Demo</A>');
      AddFooter(WebPage);
      Response.Content :=  WebPage.Text;
      Handled := True;
    end;
  finally
    WebPage.Free;
  end;
end;
```

You will notice that in each instance where a Web page is generated, we pass `WebPage` to the `AddHeader()` and `AddFooter()` procedures. The introduction page is straightforward enough. It simply contains a link to the `TWebAction`, `waUserName`. For information on `TWebAction`, see Chapter 31.

Obtaining and Verifying the User Login Name

Figure 36.5 shows the page generated by `TDDGBugsDataModule.wbdpBugswaUserNameAction()` (see Listing 36.4). This is basically an HTML form used to obtain the username. This page invokes the `TDDGBugsDataModule.wbdpBugswaVerifyUserNameAction()` event handler (see Listing 36.5).

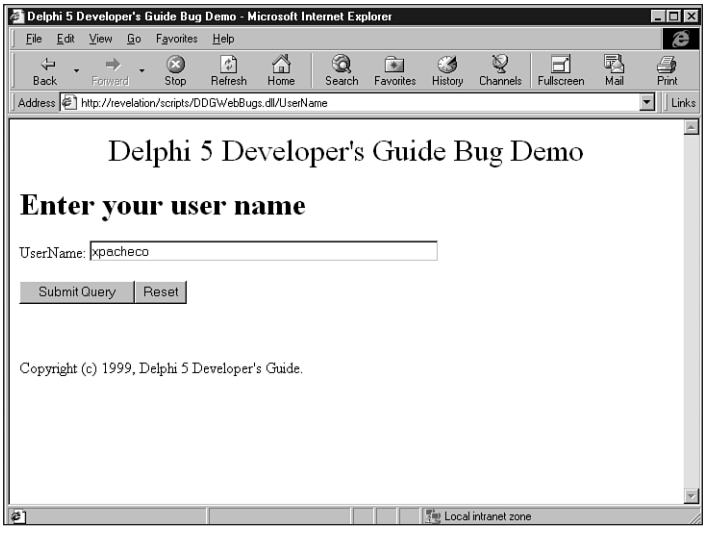

FIGURE 36.5
Obtaining the username.

LISTING 36.4 `TDDGBugsDataModule.wbdpBugswaUserNameAction()` Displays the Username Retrieval Page

```
procedure TDDGBugsDataModule.wbdpBugswaUserNameAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
// This page prompts the user for the username.
var
  WebPage: TStringList;
begin
  WebPage := TStringList.Create;
  try
    AddHeader(WebPage);
    with WebPage do
    begin
      Add('<BODY>');
      Add('<H1>Enter your user name</H1>');
      Add('<FORM action="../DDGWebBugs.dll/VerifyUserName" method="GET">');
      Add('<p>UserName: <INPUT type="text" name="UserName" maxlength="30"
➥size="50"></P>');
      Add('<p><INPUT type="SUBMIT"><INPUT type="RESET"></p>');
      Add('</FORM>');
      AddFooter(WebPage);
      Response.Content :=  WebPage.Text;
      Handled := True;
    end;
  finally
    WebPage.Free;
  end;
end;
```

LISTING 36.5 `TDDGBugsDataModule.wbdpBugswaVerifyUserNameAction()` Verifies the Username

```
procedure TDDGBugsDataModule.wbdpBugswaVerifyUserNameAction(
  Sender: TObject; Request: TWebRequest; Response: TWebResponse;
  var Handled: Boolean);
{ This page takes the name entered by the user. The information is saved
  and passed back to the client as a cookie. Additional information is also
  passed back as a cookie that will be used later for adding bugs from
  the Web. }

var
  WebPage: TStringList;
  CookieList: TStringList;
  UserName: String;
```

continues

LISTING 36.5 Continued

```
  UserFName,
  UserLName: String;
  UserID: Integer;
  ValidLogin: Boolean;

procedure BuildValidLoginPage;
begin
  AddHeader(WebPage);
  with WebPage do
  begin
    Add('<BODY>');
    Add(Format('<H1>User name, %s verified. User ID is: %d</H1>',
      [Request.QueryFields.Values['UserName'], UserID]));
    Add('<BR><BR><A href="../DDGWebBugs.dll/BrowseBugs">Browse Bug List</A>');
    Add('<BR><A href="../DDGWebBugs.dll/GetBugInfo">Add a New Bug</A>');
    AddFooter(WebPage);
  end;
end;

procedure BuildInValidLoginPage;
begin
  AddHeader(WebPage);
  with WebPage do
  begin
    Add('<BODY>');
    Add(Format('<H1>User name, %s is not a valid user.</H1>',
      [Request.QueryFields.Values['UserName']]));
    AddFooter(WebPage);
  end;
end;

begin

  UserName := Request.QueryFields.Values['UserName'];

  // The login will be valid if the username exists in the Users.db.
  ValidLogin := tblUsers.Locate('UserName', UserName, []);

  WebPage := TStringList.Create;
  try

    if ValidLogin then
    begin

      // Retrieve the UserID and the user's first and last name
      UserID := tblUsers.FieldByName('UserID').AsInteger;
```

```
      UserFName := tblUsers.FieldByName('UserFirstName').AsString;
      UserLName := tblUsers.FieldByName('UserLastName').AsString;

      CookieList := TSTringList.Create;
      try

        // Store the user's information as cookies.
        CookieList.Add('UserID='+IntToStr(UserID));
        CookieList.Add('UserName='+UserName);
        CookieList.Add('UserFirstName='+UserFName);
        CookieList.Add('UserLastName='+UserLName);

        Response.SetCookieField(CookieList, '', '', Now + 1, False);
      finally
        CookieList.Free;
      end;
      BuildValidLoginPage;
    end
    else begin
      UserID := -1;
      BuildInvalidLoginPage;
    end;

    Response.Content :=  WebPage.Text;
    Handled := True;

  finally
    WebPage.Free;
  end;

end;
```

`WbdpBugswaVerifyUserNameAction()` performs several actions. First, it verifies that the username entered represents a valid user in the `tblUsers` table. If the username is valid, the `BuildValidLoginPage()` procedure is called; otherwise, `BuildInvalidLoginPage()` is called.

If the logon is valid, the user's first and last names and user ID is retrieved from `tblUsers`. Then, these items are returned as cookies back to the client. Future requests to the Web bug server will pass these values back to the server. We will use these values in generating other pages. Finally, `BuildValidLoginPage()` is called. It constructs a page containing links for browsing bugs or adding new bugs. If the login is invalid, `BrowseInvalidLoginPage()` is called. It simply presents a message indicating the invalid login.

Assuming the user has entered a valid login, he or she has the option to browse bugs or enter a new bug.

Browsing Bugs

If the user chooses to browse bugs, he or she is presented with a page that provides the options for browsing all bugs in the database or just browsing those bugs he or she has entered. This page is constructed in `TDDGBugsDataModule.wbdpBugswaBrowseBugsAction()` and is shown in Listing 36.6.

LISTING 36.6 `TDDGBugsDataModule.wbdpBugswaBrowseBugsAction()` Displays Browsing Options for the User

```
procedure TDDGBugsDataModule.wbdpBugswaBrowseBugsAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
{ This page gives the user the option of browsing all bugs or just bugs
  entered by him/her. }
var
  WebPage: TStringList;
begin
  WebPage := TStringList.Create;
  try
    AddHeader(WebPage);
    with WebPage do
    begin
      Add('<BODY>');
      Add('<H1>Browse Option</H1>');
      Add('<BR><BR><A href="../DDGWebBugs.dll/BrowseAllBugs">
➥Browse All Bugs</A>');
      Add('<BR><A href="../DDGWebBugs.dll/BrowseYourBugs">
➥Browse Your Bugs</A>');
      AddFooter(WebPage);
      Response.Content :=  WebPage.Text;
      Handled := True;
    end;
  finally
    WebPage.Free;
  end;

end;
```

Browsing All Bugs

The option to browse all bugs invokes the `TDDGBugsDataModule.wbdpBugswaBrowseAllBugsAction()` event handler, as shown in Listing 36.7.

LISTING 36.7 `TDDGBugsDataModule.wbdpBugswaBrowseAllBugsAction()` Displays All Bugs in the System

```
procedure TDDGBugsDataModule.wbdpBugswaBrowseAllBugsAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
{ This page prepares the TPageProducer component for browsing all bugs.
  The standard header and footer is applied to this page, but a tag is
  used to add the table to the page.   }
var
  WebPage: TStringList;
begin
  WebPage := TStringList.Create;
  try
    AddHeader(WebPage);
    WebPage.Add('<BODY>');
    WebPage.Add('<H1>Browsing all Bugs</H1>');
    WebPage.Add('<#TABLE>');
    AddFooter(WebPage);

    pprdBugs.HTMLDoc.Clear;
    pprdBugs.HTMLDoc.AddStrings(WebPage);

    { As a result of the line below, the OnHTMLTag event handle for
      pprdBugs will be invoked. }
    Response.Content := pprdBugs.Content;

    Handled := True;
  finally
    WebPage.Free;
  end;
end;
```

This event handler makes use of the `TPageProducer` component `pprdBugs`. The functionality needed from this component is its capability to use tags within the HTML content. In particular, we want to use the `#TABLE` tag. We have added the standard header and footer to the Web page. However, instead of assigning `WebPage` to `Response.Content`, we assign `WebPage` to the `pprdBugs.HTMLDoc` property. Then, we assign `pprdBugs.Content` to `Response.Content`. This causes the `pprdBugs.OnHTMLTag` event to be invoked. This event, `TDDGBugsDataModule.pprdBugsHTMLTag()`, is shown in Listing 36.8.

LISTING 36.8 `TDDGBugsDataModule.pprdBugsHTMLTag()` Assigns the Table to the Tag

```
procedure TDDGBugsDataModule.pprdBugsHTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
```

continues

LISTING 36.8 Continued

```
  if Tag = tgTable then begin
    with dstpBugs do
    begin
      DataSet.Close;
      DataSet.Open;
      ReplaceText := dstpBugs.Content;
    end;
  end;
end;
```

This simple event handler assigns the `dstpBugs.Content` property, which refers to the table, to the `pprdBugs.ReplaceText` property, which will replace the `#TABLE` tag with the table contents. The resulting page is shown in Figure 36.6. It displays the bugs entered by all users.

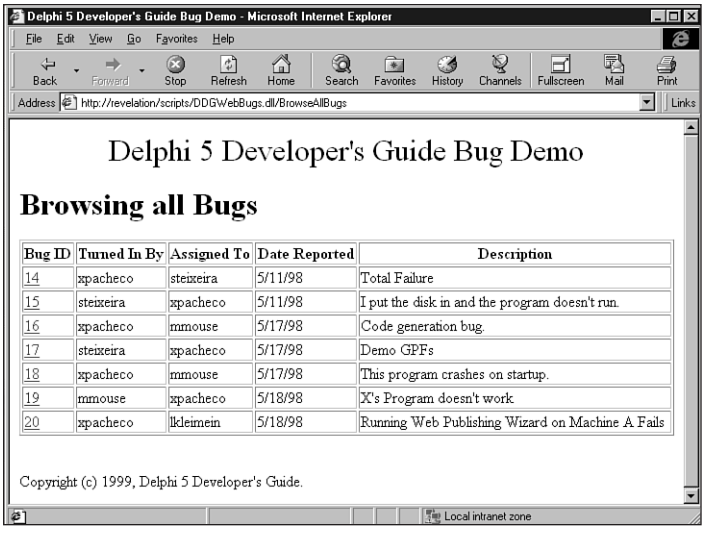

FIGURE 36.6
A list of bugs entered by all users.

Browsing User-Entered Bugs

If the user chooses to browse his or her own bugs, a page containing a table with only the bugs he or she has entered is presented to the user. The `TDDGBugsDataModule.wbdpBugswaBrowseYourBugsAction()` event handler constructs this page (see Listing 36.9).

LISTING 36.9 `TDDGBugsDataModule.wbdpBugswaBrowseYourBugsAction()` Displays Only the User's Bugs

```
procedure TDDGBugsDataModule.wbdpBugswaBrowseYourBugsAction(
  Sender: TObject; Request: TWebRequest; Response: TWebResponse;
  var Handled: Boolean);
{ This page prepares the TPageProducer component for browsing bugs which
  belong to the user. The standard header and footer is applied to this page,
  but a tag is used to add the table to the page.   }
var
  WebPage: TStringList;
  UserID: Integer;
  UserFName,
  UserLName: String;
begin
  WebPage := TStringList.Create;
  try
    AddHeader(WebPage);
    WebPage.Add('<BODY>');

    // Retrieve the user ID which is stored in the cookie.
    UserID    := StrToInt(Request.CookieFields.Values['UserID']);
    UserFName := Request.CookieFields.Values['UserFirstName'];
    UserLName := Request.CookieFields.Values['UserLastName'];

    WebPage.Add(Format('<H1>Browsing Bugs Entered by %s %s</H1>',
      [UserFName, UserLName]));
    WebPage.Add('<#TABLE>');
    pprdBugs.HTMLDoc.Clear;
    pprdBugs.HTMLDoc.AddStrings(WebPage);

    AddFooter(WebPage);

    // Make sure the table is now filtered by the UserID
    FLoginUserID := UserID;
    FilterOnUser := True;

    Response.Content := pprdBugs.Content;

    Handled := True;
  finally
    WebPage.Free;
  end;

end;
```

As was the case with the event handler for browsing all bugs, the standard header and footer need to be added to this page. Also, the `UserID`, `UserFirstName`, and `UserLastName` cookies are retrieved from the `Request.CookieFields` property. `UserFirstName` and `UserLastName` are used to display the user's name on the Web page. `UserID` is assigned `FLoginUserID`. Then the `FilterOnUser` property is set to `True`. If you recall from the previous chapter, by setting the `FilterOnUser` property to `True`, its `SetFilterOnUser()` writer method is invoked, which in turn sets `tblBugs.Filtered` to `True`. This causes the `OnFilterRecord` event handler for `tblBugs`, `tblBugsFilterRecord()`, to be called for each record in the data set. This event executes the following line of code:

```
Accept := tblBugs.FieldByName('UserID').AsInteger = FLoginUserID;
```

You can see that the filter applied depends on the value contained in the `FLoginUserID` field. This explains why the value of `UserID` from the cookie field needs to be assigned to `FLoginUserID`.

Finally, the `pprdBugs.Content` property is assigned to `Response.Content`. Again, this will cause the `pprdBugs.OnHTMLTag` event to be invoked.

Formatting Table Cells and Displaying Bug Detail

`DstpBugs` contains the `OnFormatCell` event handler `TDDGBugsDataModule.dstpBugsFormatCell()`. This event handler converts the displayed bug ID to an HTML link, which displays the detail information for that bug. `TDDGBugsDataModule.wbdpBugswaRetrieveBugAction()` is the event handler that actually displays this bug information. Both these event handlers are shown in Listing 36.10.

LISTING 36.10 The Event Handlers for Displaying Bug Detail

```
procedure TDDGBugsDataModule.dstpBugsFormatCell(Sender: TObject; CellRow,
  CellColumn: Integer; var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
{ Convert the BugID cell of the table to a link which invokes the page to
  display the bug detail. }
begin
  if (CellColumn = 0) and not (CellRow = 0) then
    CellData := Format('<A href="../DDGWebBugs.dll/RetrieveBug?
➥BugID=%s">%s</A>',
      [CellData, CellData]);
end;

procedure TDDGBugsDataModule.wbdpBugswaRetrieveBugAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
{ View the bug detail information. }
var
  BugID: Integer;
```

```
  WebPage: TStringList;

procedure GetBug;
begin
  if tblBugs.Locate('BugID', BugID, []) then
    with tblBugs do
    begin
      WebPage.Add(Format('Bug ID:        %d', [BugID]));
      WebPage.Add(Format('<BR>Reported By:   %s',
         [FieldByName('UserNameLookup').AsString]));
      WebPage.Add(FormatDateTime('"<BR>Reported On:"   mmm dd, yyyy',
        FieldByName('WhenReported').AsDateTime));
      WebPage.Add(Format('<BR>Assigned To:   %s',
         [FieldByName('AssignedToLookup').AsString]));
      WebPage.Add(Format('<BR>Status:        %s',
         [FieldByName('StatusTitle').AsString]));
      WebPage.Add(Format('<BR>Summary:       %s',
         [FieldByName('SummaryDescription').AsString]));
      WebPage.Add(Format('<BR>Details:       %s',
         [FieldByName('Details').AsString]));
      WebPage.Add('<BR>');
      WebPage.Add('<BR>');

      GetActions(WebPage);
    end;
end;

begin
  BugID := StrToInt(Request.QueryFields.Values['BugID']);

  WebPage := TStringList.Create;
  try
    AddHeader(WebPage);
    with WebPage do
    begin
      Add('<BODY>');
      Add('<H1>Bug Detail</H1>');
      GetBug;
      AddFooter(WebPage);
      Response.Content :=  WebPage.Text;
      Handled := True;
    end;
  finally
    WebPage.Free;
  end;

end;
```

Adding a New Bug

The user has the option of adding a new bug to the database. The following sections discuss the pages that retrieve the bug data from the user and display the bug information back to the user once the bug has been entered.

Retrieving the Bug Data

The event handler `TDDGBugsDataModule.wbdpBugswaGetBugInfoAction()`, shown in Listing 36.11, generates the page used to retrieve the new bug information from the user. This page basically creates an HTML form that contains the appropriate controls to allow the user to enter the proper bug information. Figure 36.7 shows the resulting page from this event handler.

LISTING 36.11 `TDDGBugsDataModule.wbdpBugswaGetBugInfoAction()` Displays the Bug Detail Entry Page to the User

```
procedure TDDGBugsDataModule.wbdpBugswaGetBugInfoAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
{ Prepares the page to retrieve new bug information from the user. }
var
  WebPage: TStringList;

procedure AddAssignToNames;
{ Adds a drop down list to the HTML Page of Assign to users }
begin

  WebPage.Add('<BR>Assign To:');
  WebPage.Add('<BR><SELECT name="AssignTo"><BR>');

  with tblUsers do
  begin
    First;
    while not Eof do
    begin
      WebPage.Add(Format('<OPTION>%s %s - %s',
        [FieldByName('UserFirstName').AsString,
        FieldByName('UserLastName').AsString,
        FieldByName('UserName').AsString]));
      tblUsers.Next;
    end;
    WebPage.Add('</SELECT>');
  end;
end;

procedure AddStatusTitles;
```

```
{ Adds a drop down list to the HTML Page of bug status items }
begin
  WebPage.Add('<BR>Status:');
  WebPage.Add('<BR><SELECT name="Status"><BR>');

  with tblStatus do
  begin
    First;
    while not Eof do
    begin
      WebPage.Add(Format('<OPTION>%s', [FieldByName('StatusTitle').AsString]));
      tblStatus.Next;
    end;
    WebPage.Add('</SELECT>');
  end;
end;

begin
  WebPage := TStringList.Create;
  try
    AddHeader(WebPage);
    with WebPage do
    begin
      Add('<BODY>');
      Add('<H1>Add New Bug</H1>');
      Add('<FORM action="../DDGWebBugs.dll/AddBug"
➥method="GET">');
      Add('<BR>Summary Description:<BR><INPUT type="text"
➥name="Summary" maxlength="100" size="50">');
      Add('<BR>Details:<BR><TEXTAREA name="Details"
➥rows=5 cols=50> </TEXTAREA>');

      AddAssignToNames;
      AddStatusTitles;

      Add('<p><INPUT type="SUBMIT"><INPUT type="RESET"></p>');
      Add('</FORM>');
      AddFooter(WebPage);
      Response.Content :=  WebPage.Text;
      Handled := True;
    end;
  finally
    WebPage.Free;
  end;
end;
```

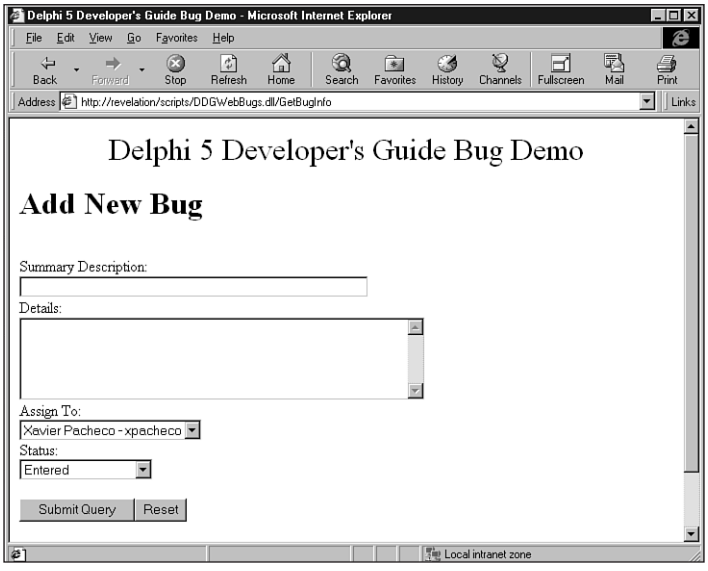

FIGURE 36.7
The bug-entry page.

The two helper functions, `AddAssignToNames()` and `AddStatusTitle()`, create combo boxes from which the user can select values for the bug. Unlike using Delphi data-aware controls that can automatically assign the selected lookup values to the new record, this assignment has to be made manually, as you will see in the event handler that adds the new bug to the database.

Verifying Bug Insertion

The event handler `TDDGBugsDataModule.wbdpBugswaAddBugAction()` is shown in Listing 36.12.

LISTING 36.12 `TDDGBugsDataModule.wbdpBugswaAddBugAction()` Adds a New Bug to the Table

```
procedure TDDGBugsDataModule.wbdpBugswaAddBugAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
{ Adds the Bug to the database. Uses the cookies returned by the client
  to display information about the user. }
var
  SummaryStr,
  DetailsStr,
  AssignToStr,
  StatusStr: String;
  WebPage: TStringList;
  UserID: Integer;
```

```
  UserName: String;
  UserFName,
  UserLName: String;
  AssignedToUserName: String;
  PostSucceeded: boolean;

function GetAssignedToID: Integer;
var
  PosIdx: Integer;
begin
  PosIdx := Pos('-', AssignToStr);
  AssignedToUserName := Copy(AssignToStr, PosIdx+2, 100);
  tblUsers.Locate('UserName', AssignedToUserName, []);
  Result := tblUsers.FieldByName('UserID').AsInteger;
end;

function GetStatusID: Integer;
begin
  tblStatus.Locate('StatusTitle', StatusStr, []);
  Result := tblStatus.FieldByName('StatusID').AsInteger;
end;

procedure DoPostSuccessPage;
begin
  with WebPage do
  begin
    Add(Format('<H1>Thank you %s %s, your bug has been added.</H1>',
        [UserFName, UserLName]));
    Add(FormatDateTime('"<BR><BR>Bug Entered on:"  mmm dd, yyyy', Date));
    Add(Format('<BR>Bug Assigned to: %s', [AssignedToUserName]));
    Add(Format('<BR>Details: %s', [DetailsStr]));
    Add(Format('<BR>Status: %s', [StatusStr]));
  end;
end;

procedure DoPostFailPage;
begin
  WebPage.Add('<BR>Bug Entry failed.');
end;

begin

  // Retrieve the fields inserted.
  SummaryStr  := Request.QueryFields.Values['Summary'];
  DetailsStr  := Request.QueryFields.Values['Details'];
  AssignToStr := Request.QueryFields.Values['AssignTo'];
```

continues

LISTING 36.12 Continued

```
StatusStr   := Request.QueryFields.Values['Status'];


// Retrieve the cookie fields.
UserID   := StrToInt(Request.CookieFields.Values['UserID']);
UserName := Request.CookieFields.Values['UserName'];
UserFName := Request.CookieFields.Values['UserFirstName'];
UserLName := Request.CookieFields.Values['UserLastName'];

// Necessary for the AfterInsert event handler.
FLoginUserID   := UserID;
FLoginUserName := UserName;

InsertBug;
try
  tblBugs.FieldByName('SummaryDescription').AsString := SummaryStr;
  tblBugs.FieldByName('WhenReported').AsDateTime := Date;
  tblBugs.FieldByName('Details').AsString := DetailsStr;
  tblBugs.FieldByName('AssignedToUserID').AsInteger := GetAssignedToID;
  tblBugs.FieldByName('StatusID').AsInteger := GetStatusID;
  tblBugs.Post;
  PostSucceeded := True;

except
  tblBugs.Cancel;
  PostSucceeded := False;
end;

WebPage := TStringList.Create;
try
  AddHeader(WebPage);
  with WebPage do
  begin
    Add('<BODY>');

    if PostSucceeded then
      DoPostSuccessPage
    else
      DoPostFailPage;

    AddFooter(WebPage);
    Response.Content :=  WebPage.Text;
    Handled := True;
  end;
```

```
  finally
    WebPage.Free;
  end;

end;
```

This event handler first retrieves all the values entered by the user from the bug-entry page shown in Figure 36.7. It also retrieves the cookie fields entered previously. The three lines of code

```
// Necessary for the AfterInsert event handler.
FLoginUserID   := UserID;
FLoginUserName := UserName;
```

are required for the `AfterInsert` event handler for `tblBugs`, which performs as follows:

```
tblBugs.FieldByName('UserID').AsInteger := FLoginUserID;
tblBugs.FieldByName('UserNameLookup').AsString := FLoginUserName;
```

Finally, the new bug is inserted into `tblBugs`. If the insertion succeeds, the Web page is constructed by calling `DoPostSuccessPage()`; otherwise, `DoPostFailPage()` is called. `DoPostSuccessPage()` simply presents the bug data back to the user, whereas `DoPostFailPage()` displays a failure notification.

Recall that data-aware lookup controls are not used to obtain valid entries for the `AssignToUserID` and `StatusID` fields for `tblBugs`. Our bug-entry page provides the user with the strings that represent these items in the drop-down combo boxes. In order to add the proper lookup index values to `tblBugs`, a search is performed on the strings selected by the user against both `tblUsers` and `tblStatus`. Note that a bit of string manipulation is required for the `AssignToUserID` field in order to extract the proper string with which to perform the search (see the `GetAssignToID()` method).

Summary

This chapter covered deploying Web database applications. In this chapter, we demonstrated how, if properly designed, an existing application can be deployed to the Web with few modifications to the existing code (with the exception of adding code specific to the Web). In fact, most of what we presented here has more to do with the construction of HTML documents than with database manipulation. You might consider modifying this demo to extend its functionality as well as moving the HTML construction code to actual HTML files.