# Building WebSnap Applications

*by Nick Hodges*

CHAPTER

# 23

## IN THIS CHAPTER

Delphi 6 introduces a new Web application framework called WebSnap that brings the strengths of *Rapid Application Development (RAD)* to Web development. Building on WebBroker and InternetExpress, WebSnap is a big leap forward for Delphi developers who want to use their favorite tool to build Web applications. It provides all the standard nuts and bolts for Web applications, including session management, user login, user preference tracking, and scripting. Naturally, Delphi 6 brings RAD to Web site development, making building robust, dynamic, database-driven Web applications easy and fast.

# WebSnap Features

WebSnap isn't a totally new technology, and it doesn't leave behind your WebBroker and InternetExpress applications. WebSnap is compatible with these two older technologies, and it is a relatively straightforward process to integrate your existing code into a new WebSnap application. WebSnap provides several features listed in the following sections.

## Multiple Webmodules

In Delphi's previous versions, WebBroker and InternetExpress applications had to do all their work in a single Web module. Multiple webmodules weren't allowed. To add datamodules, they had to be created manually at runtime, rather than automatically. WebSnap eliminates this restriction and allows any number of webmodules and datamodules to be part of a Web application. WebSnap is based on multiple modules, and each module represents a single Web page. This allows different developers to work on different portions of the application without having to worry about modifying each other's code.

## Server-side Scripting

WebSnap seamlessly integrates server-side scripting into your applications, and allows you to very easily build powerful scriptable objects that you can use to build and customize your applications and HTML. The `TAdapter` component and all of its descendents are scriptable components, meaning that they can be called by your server-side script and produce HTML and client-side JavaScript for your applications.

## `TAdapter` Components

`TAdapter` components define an interface between an application and the server-side scripting. Server-side script only has access to your application via adapters, ensuring that the script doesn't inadvertently change the data in an application or expose functions that aren't intended for public consumption. You can build custom `TAdapter` descendents that manage content for your specific needs, and that content can even be visible and configurable at design time. `TAdapters` can hold data and execute actions. For instance, the `TDataSetAdapter` can display

records from a dataset as well as take the normal actions on a dataset such as scroll, add, update, and delete.

## Multiple Dispatching Methods

WebSnap provides a number of ways to manage HTTP requests. You can access your Web content by page name, by `TAdapter` actions, or by simple Web action requests as WebBroker does. This gives you the power and flexibility to display your Web pages based on any number of different kinds of inputs. You might want to display a page in response to a submit button, or you might want to build a set of links into a menu based on the collection of pages in your site.

## Page Producer Components

WebBroker introduced `TPageProducer`, a component for managing HTML and inserting and updating content based on custom tags. InternetExpress advanced this notion with `TMidasPage Producers`. WebSnap advances the notion of PageProducers even further, adding a number of new and powerful controls that can access `TAdapter` content, as well as XSL/XML data. The most powerful of these new `TPageProducer` descendents is `TAdapterPageProducer`, which knows how to produce HTML based on the actions and fields of `TAdapter` components.

## Session Management

WebSnap applications contain automatic, built-in session management; now you can keep track of user's actions across multiple HTTP requests. Because HTTP is a stateless protocol, your Web applications must keep track of users by leaving something on the client that identifies each user. Normally this is done with cookies, URL references, or hidden field controls. WebSnap provides seamless session support that makes tracking users very easy. WebSnap does this via its `SessionsService` component. The `SessionsService` component seamlessly maintains a session identification value for each user, making it a simple task to keep track of each user as she makes individual requests. This is normally a difficult service to manage, but WebSnap handles all the details and makes the session information available both in server-side script and the Web application code itself.

## Login Services

Your Web applications will likely need security to be implemented, requiring users to log in to the given application. WebSnap automates this process by providing a specialized login adapter component. This component contains the functions needed to properly query and authenticate users according to the application's chosen security model. It gathers login information, and in conjunction with WebSnap's session management, provides current login

credentials for each request. The login components also automate login validation and login expiration. Throughout your application, users who try to access unauthorized pages can be automatically referred to the login page.

## User Tracking

The most common function that session tracking provides is the ability to keep track of your users and their preferences for your application. WebSnap provides components that allow you to easily track user information and display it on your site. You can store user login information, and then retrieve user information based on that. You can maintain user access rights and site preferences, as well as things such as shopping cart information.

## HTML Management

Often in a dynamic Web application, keeping track of and managing HTML can be difficult. HTML content can reside in any number of places such as files and resources, or they can be dynamically generated. WebSnap provides a means for you to manage this process with its file location services.

## File Uploading Services

Managing the uploading of files usually requires a lot of custom code. WebSnap provides a simple adapter solution that manages the multipart forms needed to upload files. You can provide file upload capability in your WebSnap application quickly and easily using the built-in functionality of the `TAdapter` component.

# Building a WebSnap Application

As always, the best way to learn about the new technology in Delphi is to try it out. We'll start by building the `"Hello World"` version of a WebSnap application.

## Designing the Application

First, you'll want to add the WebSnap toolbar to the IDE, so right-click on the speedbutton area of the IDE title bar, and select the Internet toolbar (see Figure 23.1). This adds a toolbar to the IDE main window that makes creating WebSnap applications and adding forms and webmodules easy.

Next, click the speedbutton with the hand holding the globe, and you will see dialog box shown in Figure 23.2.

**FIGURE 23.1**
*Internet toolbar.*

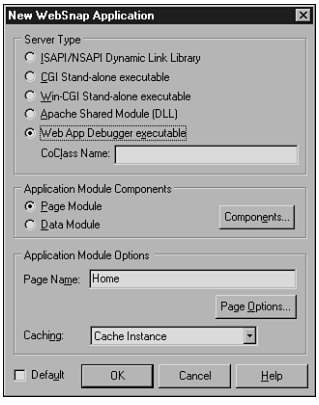


**FIGURE 23.2**
*The New WebSnap Application dialog box.*



The dialog box in Figure 23.2 gives you a number of options for setting up your WebSnap application. The first is the type of server that your application is going to run on. You are given five choices:

- ISAPI/NSAPI Dynamic Link Library—This option produces a project that runs under IIS (or under Netscape servers with the appropriate ISAPI adapter installed). The project produces a DLL when compiled, and runs in the same memory space as the Web server. The most common Web server to run ISAPI applications is Microsoft's Internet Information Server, although other Web servers can run ISAPI DLLs.
- CGI Standalone executable—This option creates a project that produces a console executable that reads and writes from the standard input and output ports. It conforms to the CGI specification. Almost all Web servers support CGI.
- Win-CGI Standalone executable—This option produces a Win-CGI project that communicates with a Web server via text-based INI files. Win-CGI is very uncommon and not recommended.

- Apache Shared Module (DLL)—This option produces a project that will run in the Apache Web server. For more information about Apache, see `http://www.apache.org`.
- Web App Debugger Executable—If you select this option, you get an application that will be run by Delphi's Web App Debugger (see Figure 23.3). Your Web application will be an out-of-process COM server, and the Web App Debugger will control and run the application. This type of Web application will allow you to use the full power of Delphi's debugger when debugging it. This means no more hassling with Web servers, turning them on and off in order to load and unload your applications. Instead, debugging your application will be fast and easy.

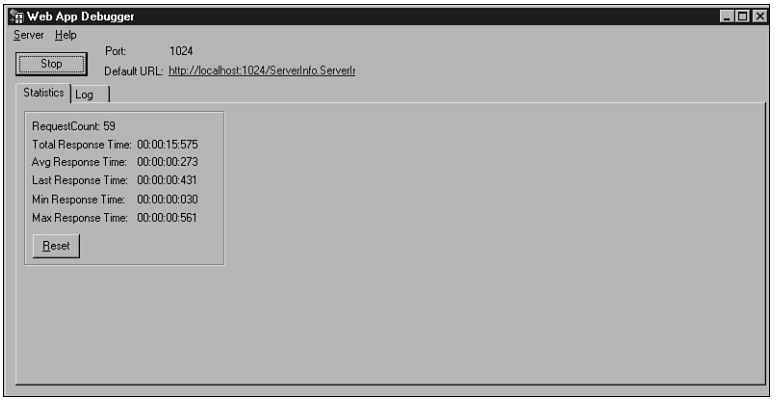


**FIGURE 23.3**
*The Web App Debugger application greatly simplifies debugging your Web applications.*

**NOTE**

The WebApp Debugger can be accessed via the Tools menu in the IDE. In order to work properly, you need to register the application found in the `<Delphi Dir>\bin` directory called `serverinfo.exe`. All you need to do to register it is run it once, and it will register itself. The Web App Debugger is a COM-based application that acts as a Web server to your testing applications. When you create a Web App Debugger Application, your new project will contain a form and a Web module. The form acts as a placeholder for the COM server, and running the application once will register it. After that, the Web App Debugger will control it via the Web browser, and will serve your application in the browser. Because the application is a Delphi executable and not a Web server extension, you can set a breakpoint in it and run it in the Delphi IDE. Then, when you access it through the browser, Delphi's debugger will take over when your breakpoints are reached, and you can debug the application normally.

*continues*

To access your application via the browser, run the Web App Debugger, and click on the hyperlink labeled Default URL. This will bring up a Web application that lists all the applications registered with the server. You can then select your application and run it. The View Details option will allow you to see more information about the different applications, and to clean them out of the registry when they are no longer needed. Be careful, though, not to delete the `ServerInfo` application; otherwise, you'll have to go back and register it again.

For the sample application that you will build here, select the Web App Debugger option. This will allow you to debug the application as you build it.

The next option in the wizard allows you to select the type of module you want and the different components that will be included. If you choose the Page Module option, you will get a Web module that represents a page in your application. If you choose the Data Module option, you will get a datamodule that can be used in a WebSnap application. It can perform the same function as datamodules do in traditional client/server applications. For this application, select the Page Module option.

Next, click on the Components button, and you'll see the dialog box shown in Figure 23.4.



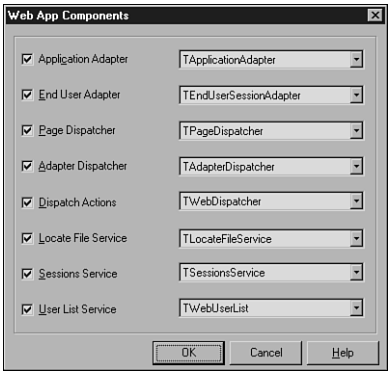


**FIGURE 23.4**
*The Web App Components dialog box allows you to select the components that will be included in your new module.*

You have the choice of the following components listed:

- Application Adapter—This component manages the fields and actions available through the Application server-side scripting object. The most common property you'll use in this component is the Title property.

- End User Adapter—This component manages the information about the current user of the application such as the session ID, username, user rights, and other customized user information. It also will manage the user's login and logout actions.
- Page Dispatcher—This component manages and dispatches HTTP requests made by page name. You can create HREF links or actions that call specific pages, and the Page Dispatcher will retrieve the proper response.
- Adapter Dispatcher—The Adapter Dispatcher handles all requests that come as a result of adapter actions. These are generally the result of an HTML form submission.
- Dispatcher Actions—This option adds a `TWebDispatcher` to your applications. Users of WebBroker will remember this component. It handles requests from the application based on URLs, just as WebBroker applications did. You can use this component to add your own custom actions to your application in the same way you did with WebBroker.
- Locate File Service—The events of this component are called whenever a Web module requires HTML input. You can add event handlers that allow you to bypass the default HTML finding mechanism and get HTML from almost any source. This component is used most often for grabbing page content and templates for building standard pages.
- SessionsService—This component manages sessions for users, allowing you to maintain state for individual users between HTTP requests. The SessionsService can store information about users and automatically expire their sessions after a certain period of inactivity. You can add any session-specific information you want to the `Session.Values` property, a string indexed array of variants. By default, the sessions are managed using cookies on the user's machine, although you could build a class to handle them some other way, such as with fat URLs or hidden fields.
- User List Service—This component maintains a list of users who are authorized to log in to the application and information about them.

**NOTE**

These options each have drop-down boxes that allow you to choose the component that will fulfill each of the preceding roles. You can create your own components that will fulfill these roles and register them with WebSnap. They will then appear as choices in this dialog box. You could, for instance, create a session component that maintains session information in a fat URL rather than with cookies.

For this example, select all the check boxes. Then, for the End User Adapter component, drop down the combo box and select `TEndUserSessionAdapter`. This component will automatically associate a session ID with an end user. Then click OK.

The next option in the wizard is the name of the page. Name this main page **Home**, and then click Page Options. You'll see dialog box shown in Figure 23.5.

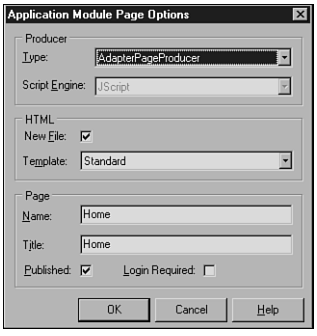


**FIGURE 23.5**
*The Application Module Page Options dialog box allows you to select the options for the page in your Web module.*

This dialog box allows you to customize the `PageProducer` component of your Web module and the HTML associated with it. It presents a number of options. The first is the type of page producer. WebSnap includes a number of standard PageProducers that can produce and manage HTML in different ways. To start with, select the default option, a plain PageProducer. You can also select the type of server-side scripting you want to use. Out of the box, Delphi supports JScript and VBScript (as well as XML/XSL, which will be discussed later.) Leave the default value of JScript here.

Each module has an HTML page associated with it. The next option allows you to select what type of HTML you want. By default, Delphi provides a Standard page with a simple scripted navigation menu on it. You can create your own HTML templates, register them with WebSnap, and then select them here. We'll look at how to do that later in this chapter. For now, leave the default value of Standard here.

Name the page **Home** (the Title is automatically filled in the same). Make sure Published is checked, and leave Login Required unchecked. A published page will show up in the list of pages in the application and can be referenced by the Pages scripting object. This is needed to create page-based menus in script using the Pages scripting object.

After you have done this, click OK, and then click OK on the main wizard. The wizard will then create your application for you, and the new Web module will look something similar to that shown in Figure 23.6.

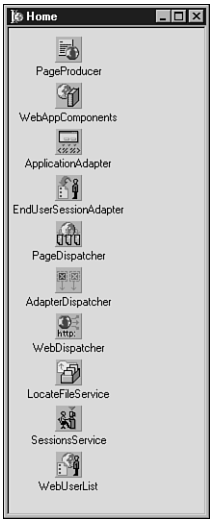


**FIGURE 23.6**
*The Web module for the demo application as created by the WebSnap Wizard.*

We haven't yet discussed the `TWebAppComponents` control. This control is the *central clearing house* for all the other components. Because many of the components in a WebSnap application work together, the `WebAppComponents` component is the one that ties them together and allows them to communicate and refer to each other. Its properties consist merely of other components that fill the specific roles discussed previously.

At this point, you should save the project. To keep consistent with the rest of the chapter, name the Web module (unit2) wmHome, name the form (Unit1) ServerForm, and name the project itself DDG6Demo.

By examining the Code Editor, you should see some new, unfamiliar features. First, notice the tabs along the bottom. Each Webmodule—because it represents a page in a Web application—has an associated HTML file that can contain server-side script. The second tab on the bottom shows this page (see Figure 23.7). Because you selected the Standard HTML page template in the wizard, the HTML contains server-side script that will greet the user if she is logged in, and will provide a basic navigation menu that will be automatically built based on all the published pages in the application. As pages get added to this demo application, this menu will grow larger and will allow users to navigate to each of the pages. The default HTML code is shown in Listing 23.1.

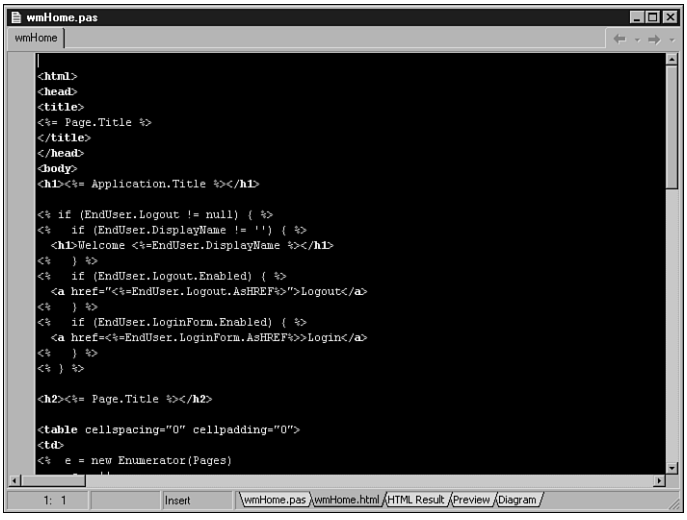


**FIGURE 23.7**
*The HTML page associated with the Web module.*



**LISTING 23.1** Default HTML Code

```
<html>
<head>
<title>
<%= Page.Title %>
</title>
</head>
<body>
<h1><%= Application.Title %></h1>

<% if (EndUser.Logout != null) { %>
<%   if (EndUser.DisplayName != '') { %>
  <h1>Welcome <%=EndUser.DisplayName %></h1>
<%   } %>
<%   if (EndUser.Logout.Enabled) { %>
  <a href="<%=EndUser.Logout.AsHREF%>">Logout</a>
<%   } %>
<%   if (EndUser.LoginForm.Enabled) { %>
  <a href=<%=EndUser.LoginForm.AsHREF%>>Login</a>
<%   } %>
<% } %>
```

**LISTING 23.1** Continued

```
<h2><%= Page.Title %></h2>

<table cellspacing="0" cellpadding="0">
<td>
<%  e = new Enumerator(Pages)
     s = ''
     c = 0
     for (; !e.atEnd(); e.moveNext())
     {
       if (e.item().Published)
       {
         if (c>0) s += ' | '
         if (Page.Name != e.item().Name)
           s += '<a href="' + e.item().HREF + '">' + e.item().Title + '</a>'
         else
           s += e.item().Title
         c++
       }
     }
     if (c>1) Response.Write(s)
%>
</td>
</table>

</body>
</html>
```

This code contains both normal HTML tags as well as server-side JScript.

You should also note that the HTML is syntax-highlighted in the IDE. (You can set the colors to be used in Tools, Editor Options, Color property page.) In addition, you can set your own external HTML editor such as HomeSite and access it via the IDE as well. Set the HTML Editor in Tools, Environment Options, Internet. Select HTML in the listview, and then click Edit. From there, select the appropriate edit action to use your external editor. Then, when you right-click on the HTML page in the Code Editor, you can select the HTML Editor option and call up your editor.

In addition to the HTML viewing tab, the next tab shows the HTML that results from the script being run. The following tab shows a preview of the HTML in an Internet Explorer window. Do note that not all the script will execute and display in this view because some of the code relies on runtime values. However, you can at least get an idea what the page will look like without having to run it in the browser.

## Adding Functionality to the Application

Now let's add a little code and make the application do something. First, go to the Home Web module and select the Application adapter. Set the `ApplicationTitle` property to Delphi Developers Guide 6 WebSnap Demo Application. Note that this will immediately show up in the preview tab because the HTML contains the following server-side script as the first thing in the `<BODY>` section:

```
<h1><%= Application.Title %></h1>
```

This causes the Application scripting object to display the value for `ApplicationTitle` in the HTML.

Next, go to the Code Editor, and select the HTML page for the Home Module. Then move the cursor down below the `</table>` tag near the bottom and add a pithy description of the page, which welcomes the user. The code on the CD-ROM has such an entry, adding the following:

```
<P>
<FONT SIZE="+1" COLOR="Red">Welcome to the Delphi 6 Developers Guide WebSnap
➥ Demonstration Application!</FONT>
<P>
This application will demonstrate many of the new features in Delphi 6 and
➥WebSnap. Feel free to browse around and look at the code involved. There is
➥a lot of power, and thus a lot to learn, in WebSnap, so take your time and
don't try to absorb it all at once.
<P>
```

This new code of course immediately shows up in the HTML Preview panel as well.

Next, just to prove that you are actually building a browser application, run the project. The first thing you will see is a blank form. This is the COM server. You can shut it down once it runs, and then start up the Web App Debugger from the Tools menu. After you have done that, click on the Default URL hyperlink (it will be called `DDG6DemoApp.DDG6TestApp`), find the application in the list box in your browser, and click the Go button. Your browser should show your page as illustrated in Figure 23.8.

As you can see, it really is a Web application!

## Navigation Menu Bar

Now, you'll add another page that demonstrates the navigation menu. Go to the IDE's main menu bar, and select the second toolbutton on the Internet menu, the one with the little globe and the sheet of paper. This will bring up the New WebSnap Page Module Wizard, which is similar to the dialog box you saw as part of the main wizard. Leave all the options with the default values, except for the Name edit box. Name the page Simple. The result is a Web module with a single PageProducer in it. Note that an HTML page is associated with this page, and it has the same code as the first page you saw. Save the unit as `wmSimple.pas`.

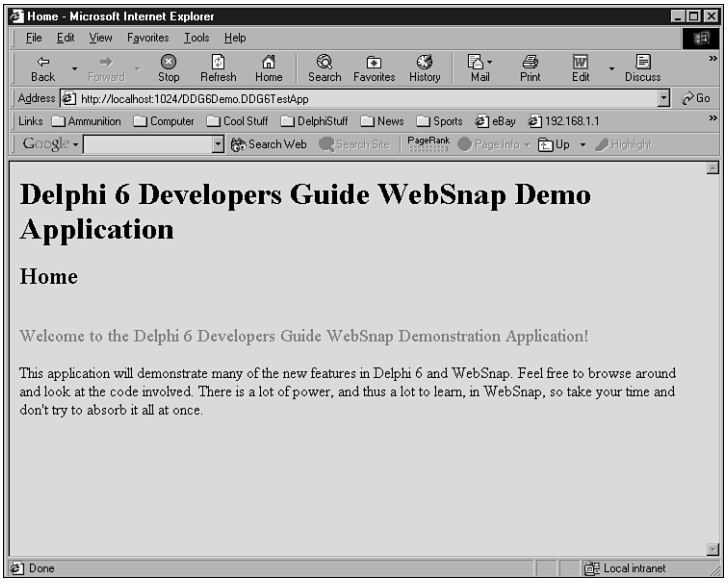


**FIGURE 23.8**
*Results of the first page added to the demo application.*

## Setting Threading and Caching Options

The New WebSnap Page Module Wizard has two options at the bottom that determine how instances of each Web module are to be handled. The first is the Creation option. Web modules can be created either On Demand or Always. Web modules created On Demand are only instantiated when a request comes in for them. Choose this option for pages that are less frequently used. Choose Always for pages that are created immediately upon application startup. The second option is the Caching Option, and this determines what happens to a Web module when it has finished servicing its request. If Cache Instance is chosen, each Web module created is cached when it is finished providing a request, and it remains in a pool of cached instances, ready to be used again. It is important to note that when it is used again, the field data will be in the same state it was in when it finished its last request. Choose Destroy Instance if you want each instance of the Web module to be destroyed upon completion instead of being cached.

Next, add some simple message in the HTML page in the same spot below the table in the standard page. Then, compile and run the application via the Web App Debugger as you did

before. If the page was there from the last time you checked it, all you need to do is click the Refresh button on your browser.

This time when you run the application, you should note that the navigation menu now appears. That menu is a result of the following server-side script:

```
<%  e = new Enumerator(Pages)
     s = ''
     c = 0
     for (; !e.atEnd(); e.moveNext())
     {
       if (e.item().Published)
       {
         if (c>0) s += ' | '
         if (Page.Name != e.item().Name)
           s += '<a href="' + e.item().HREF + '">' + e.item().Title + '</a>'
         else
           s += e.item().Title
         c++
       }
     }
     if (c>1) Response.Write(s)
%>
```

This code simply iterates over the Pages scripting object, building a menu of page names. The code makes a link if the page found isn't the current page. Thus, the current page isn't a link, and all the other page names are, no matter what the current page is. This is a rather simple menu, and of course you could write your own more sophisticated menus for your custom application.



**NOTE**

If you toggle between the two pages, you might notice that the application's form flashes in the background each time a request is made. That is because the Web App Debugger is calling the application as a COM object for each request, running the application, getting the HTTP response back, and shutting down the application.

Next, you can make part of the application restricted only to users who are logged in. First, add a page that requires a user to be logged in to see it. Click the New WebSnap Page button on the Internet toolbar, and name the page "LoggedIn." Then, select the LoginRequired check box. Click OK to create a Web page that can only be viewed by a user who is logged in. Save the page as `wmLoggedIn.pas`. Then, add some HTML code to the HTML page letting the user

know that only logged in users can view the page. The application on the CD-ROM includes the following:

```
<P>
<FONT COLOR="Green"><B>Congratulations! </B></FONT>
<BR>
You are successfully logged in!  Only logged in users are granted access
➥to this page. All others are sent back to the Login page.
<P>
```

The only difference between the LoggedIn page and the Simple page is a parameter in the code that registers the page with the application manager. Every WebSnap page has an initialization section that looks something like this:

```
initialization
  if WebRequestHandler <> nil then
  WebRequestHandler.AddWebModuleFactory(TWebPageModuleFactory.Create(TLoggedIn,
TWebPageInfo.Create([wpPublished, wpLoginRequired], '.html'),
➥crOnDemand, caCache));
```

This code registers the page with the `WebRequestHandler` object, which manages all the pages and provides their HTML content when needed. `WebRequestHandler` knows how to create, cache, and destroy instances of webmodules as needed. The preceding code is the code for the LoggedIn page, and it has the `wpLoginRequired` parameter, telling the page that only logged in users can access it. By default, the New Page Wizard adds this value, but comments it out. If you want the page to become password protected later, you can simply uncomment the parameter and recompile the application.

## Logging In

You need to create a page that lets the user log in. First, however, there is some housekeeping to do on the Home page.

First, create a new page and give it the name Login. Then, select `TAdapterPageProducer` for the page producer type. This time, however, don't publish it by deselecting the Publish check box, and obviously don't require a user to be logged in to view the login page! Deselecting the Publish option will make the page available for use, but it won't be part of the Pages scripting object, and thus it won't show up on the navigation menu. Save it as wmLogin. This time, go to the WebSnap page of the Component Palette and drop a `TLoginAdapter` component on the module.

The `TAdapterPageProducer` is a specialized PageProducer that knows how to display and handle the appropriate HTML fields and controls for a `TAdapter`. In the case of the Demo application, this `TAdapterPageProducer` is going to display the Username and Password edit boxes that the user will need to use to log in. When you begin to understand WebSnap better, you'll

quickly want to use TAdapterPageProducers in all your pages because they make it very easy to display TAdapter information, execute TAdapter actions, and build HTML forms based on TAdapter fields.

Because the TLoginFormAdapter has all the fields needed for this, creating the login page will be very easy, and done with no code at all—that's right, no code. You'll be able to add users, create a login page, and enforce the login on pages you specify, all without a single line of code.

First, to manage logins, you'll need to create some users. Go to the Home Web module and double-click on the WebUserList component. This component manages users and passwords. You can easily add users and their passwords. Click on the New button and add two different users. Add whatever passwords you want for each user. The two users on the demo application on the CD-ROM are ddg6 and user. Their passwords are the same as their usernames, as shown in Figure 23.9.

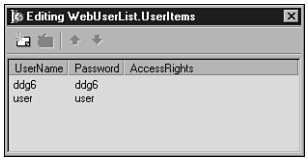


**FIGURE 23.9**
*The component editor for the* WebUserList *component with two users added.*



Select the EndUserSessionAdapter and set the LoginPage property to Login, that is, the name of the page that has the controls to log in users. Next, go back to the Login Web module and double-click on the TAdapterPageProducer component. This will bring up the Web Surface designer, shown in Figure 23.10.

Select the AdapterPageProducer in the upper right, and click the New Component button. Select AdapterForm and click OK. Then, select the AdapterForm1, and click the New Component button again. Select AdapterErrorList. Do the same for AdapterFieldGroup and AdapterCommandGroup. Then set the Adapter property for these three components to LoginFormAdapter1. Then, select the AdapterFieldGroup and add two AdapterDisplayField objects. Set the FieldName property on the first one to UserName, and the second one to Password. Select the AdapterCommandGroup, and set its DisplayComponent property to AdapterFieldGroup1. You should then have a form that looks like Figure 23.10. If you close this form, and then go to the Code Editor, you can see that the form now has the login controls in it.

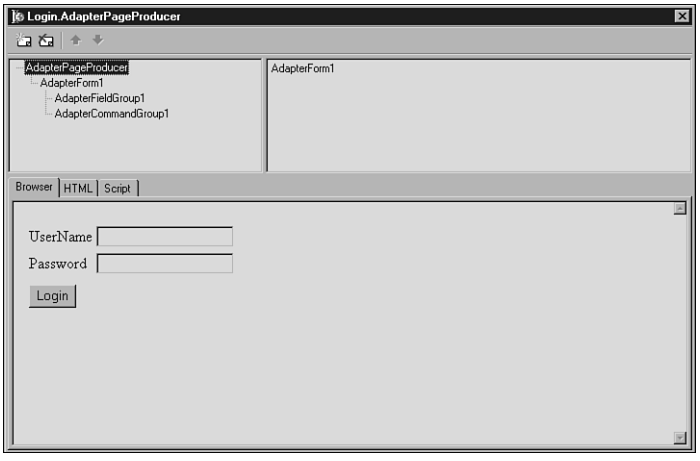


**FIGURE 23.10**
*The* `TAdapterPageProducer` *Web Surface Designer with the LoginFormAdapter components on it.*

That's all you need to do. Run the application and leave it running. Now, because you are relying on session information about the user, you need to leave the application in memory for it to remember that you are logged in. Run the application in the browser, and then try to navigate to the page that requires you to be logged in. It should take you to the Login page. Enter a valid username and password, click the login button, and you will be taken to the page asking you to log in. From now on, any page that you specify as requiring a valid login will only display if you are properly logged in. Otherwise it will send you to the login page. All that happens without writing a single line of Pascal code.

Try this as well: Log out by selecting the Logout link, and then try to login with an invalid username or password. Note that an error message is displayed. That is the `AdapterErrorList` component at work. It automatically collects login errors and displays them for you.

When you are logged in to the application and navigating around the pages in the application, you will notice that it remembers who you are and displays your login name in the heading for each page. This is a result of the following server-side script in the HTML file for the webmodules:

```
<% if (EndUser.Logout != null) { %>
<%   if (EndUser.DisplayName != '') { %>
  <h1>Welcome <%=EndUser.DisplayName %></h1>
<%   } %>
```

## Managing User Preference Data

The next thing you might want to do is to maintain some user preference information. Most dynamic, user-based applications will want to display all different types of user information ranging from items in a shopping cart to a user's color preferences. Of course, WebSnap makes this very easy. But this time, you'll actually have to write a few lines of code.

First, add another page to the application. Give it a `TAdapterPageProducer` and require the user to be logged in to view it. (By now, you should be able to do this using the toolbar and the resulting wizard.) Save the file as `wmPreferenceInput`. Add a `TAdapter` to the Webmodule. Rename the `Adapter` from `Adapter1` to `PrefAdapter`, as shown in Figure 23.11.

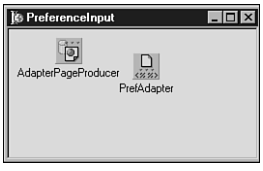


**FIGURE 23.11**
*The* `PreferenceInput` *Web module that will gather up the user's preferences.*



First, double-click on the `PrefAdapter` component, and then add two `AdapterFields` and one `AdapterBooleanField`. Name the two `AdapterFields` FavoriteMovie and PasswordHint. Name the `AdapterBooleanField` LikesChocolate. (Notice that when you rename these components, the `DisplayLabel` and `FieldName` values change as well.) You can also change the `DisplayLabel` values that make more sense in your HTML.

The `PrefAdapter` component will hold the values for these preferences, and they can be accessed from other pages. `TAdapters` are scriptable components that can hold, manage, and manipulate information for you, but doing that will require some code. Each of the three `AdapterFields` you created need to be able to retrieve their values when asked for them in script, so each has an `OnGetValue` event that does just that. Because you want this information to be persistent across requests, you'll store the information in the `Session.Values` property. The `Session.Values` variable is a string-indexed array of variants, so you can store almost anything in it, and it will maintain that information as long as the current session is active.

The `TAdapter` class also allows you to take actions on its data. Most commonly, this will take the form of a Submit button on your HTML form. Select the `PrefAdapter` component, go to the Object Inspector, and double-click on the Actions property. Add a single action and name it `SubmitAction`. Change its `DisplayLabel` property to Submit Information. Then, go to the Events page in the Object Inspector and add this code to the action's `OnExecute` event as shown in Listing 23.2.

**LISTING 23.2** `OnExecute` Handler

```
procedure TPreferenceInput.SubmitActionExecute(Sender: TObject;
  Params: TStrings);
var
  Value: IActionFieldValue;
begin

  Value := FavoriteMovieField.ActionValue;
  if Value.ValueCount > 0 then
  begin
    Session.Values[sFavoriteMovie] := Value.Values[0];
  end;

  Value := PasswordHintField.ActionValue;
  if Value.ValueCount > 0 then
  begin
    Session.Values[sPasswordHint] := Value.Values[0];
  end;

  Value := LikesChocolateField.ActionValue;
  if Value <> nil then
  begin
    if Value.ValueCount > 0 then
    begin
      Session.Values[sLikesChocolate] := Value.Values[0];
    end;
  end else
  begin
    Session.Values[sLikesChocolate] := 'false';
  end;;
end;
```

This code retrieves the values from the input fields in your HTML when the user clicks the Submit button, and puts the values in the session variable for later retrieval by the `AdapterFields`.

Of course, you need to be able to retrieve those values once they are set, so each field of the adapter will get its value back from the SessionsService object. For each field in the adapter, make the `OnGetValue` event handlers resemble the code in Listing 23.3.

**LISTING 23.3** `OnGetValue` Event Handlers

```
...
const
  sFavoriteMovie = 'FavoriteMovie';
  sPasswordHint = 'PasswordHint';
```

**LISTING 23.3** Continued

```
  sLikesChocolate = 'LikesChocolate';
  sIniFileName = 'DDG6Demo.ini';
...

procedure TPreferenceInput.LikesChocolateFieldGetValue(Sender: TObject;
  var Value: Boolean);
var
  S: string;
begin
  S := Session.Values[sLikesChocolate];
  Value := S = 'true';
end;

procedure TPreferenceInput.FavoriteMovieFieldGetValue(Sender: TObject;
  var Value: Variant);
begin
  Value := Session.Values[sFavoriteMovie];
end;

procedure TPreferenceInput.PasswordHintFieldGetValue(Sender: TObject;
  var Value: Variant);
begin
  Value := Session.Values[sPasswordHint];
end;
```



Next, you need to be able to display controls that will actually get the data from the user. You'll do that via the `TAdapterPageProducer`, just as you did with the Login page. First, double-click on the `TAdapterPageProducer`, and you will get the Web Surface designer again. Create a new `AdapterForm`, and then add an `AdapterFieldGroup`, as well as an `AdapterCommandGroup`. Set the `Adapter` property of the `AdapterFieldGroup` to `PrefAdaper`, and set the `DisplayComponent` of the `AdapterCommandGroup` to `AdapterFieldGroup`. Then, right-click on the `AdapterFieldGroup` and select Add All Fields from the menu. For each of the resulting fields, use the Object Inspector to set the `FieldName` property to the appropriate values. You can also change the `Caption` properties to more friendly values than the default. Then select the `AdapterCommandGroup`, right-click on it, and select Add All Commands from the menu. Set the `ActionName` property of the resulting `AdapterActionButton` to `SubmitAction`. Finally, set the `AdapterActionButton.PageName` property to `PreferencesPage`. (This is the page that the action will go to once it is done processing the action. You'll create that page in a minute.)

If something isn't hooked up correctly in the Web Surface Designer, you will see an error message in the Browser tab. The message will instruct you on the properties that need to be set for everything to be connected properly and for the HTML to be rendered properly.

After you have done all this, and the HTML looks right, the page is done. Now, if you run the application, you'll see an additional page on the menu. If you log in, you can see the input controls to enter your preference data. Don't click the Submit button just yet because there is no place to go.

Next, create a page to display the user preferences by using the toolbar, and then name it PreferencesPage. Publish the page and require users to be logged in to view it. (Again, the wizard can do all this for you as before.) Save the new unit as `wmPreferences`.

Then, go the HTML for the page, and in the area just below the table that holds the navigation menu, add the following script:

```
<P>
Favorite Movie: <%= Modules.PreferenceInput.PrefAdapter.FavoriteMovieField.
➥Value %>
<BR>
Password Hint:  <%= Modules.PreferenceInput.PrefAdapter.PasswordHintField.
➥Value %>

<BR>
<% s = ''
  if (Modules.PreferenceInput.PrefAdapter.LikesChocolateField.Value)
     s = 'You like Chocolate'
   else
     s = 'You do not like chocolate'
   Response.Write(s);
%>
```

Now, when you compile and run the application, you can enter your preferences and click the Submit button—the application will remember and display your preferences in the Preferences page. You can access those values in the script for any other page as well once they are set. The values are maintained between HTTP requests by the Session object and retrieved from the `Adapter` component via script.

**NOTE**

Each of the pages in a WebSnap application has an associated HTML file, as you have seen. Because these files exist outside of the application, you can edit them, save the changes, refresh the page in your browser, and see the results without recompiling your application. This means that you can update the page itself without having to take down your Web server. You can also easily experiment with your server-side script during development without having to recompile your application. Later in the chapter, you'll look at alternative ways to store and retrieve your HTML.

## Persisting Preference Data Between Sessions

There's only one problem now—the user's selections aren't persistent between sessions. The preferences are lost if the user logs out. You can make these values persist even between sessions by storing them each time the session ends and grabbing them each time a user logs in. The demo application reads any stored data in the `LoginFormAdapter.OnLogin` event, and then writes out any data in the `SessionService.OnEndSession` event. The code for those two events is shown in Listing 23.4.

**LISTING 23.4** `OnLogin` and `OnEndSession` Events

```
procedure TLogin.LoginFormAdapter1Login(Sender: TObject; UserID: Variant);
var
  IniFile: TIniFile;
  TempName: string;
begin
  // Grab session data here
  TempName := Home.WebUserList.UserItems.FindUserID(UserId).UserName;
➥  //WebContext.EndUser.DisplayName;
  Home.CurrentUserName := TempName;

  Lock.BeginRead;
  try
    IniFile := TIniFile.Create(IniFileName);
    try
      Session.Values[sFavoriteMovie] := IniFile.ReadString(TempName,
      ➥sFavoriteMovie, '');
      Session.Values[sPasswordHint] := IniFile.ReadString(TempName,
      ➥sPasswordHint, '');
      Session.Values[sLikesChocolate] := IniFile.ReadString(TempName,
      ➥sLikesChocolate, 'false');
    finally
      IniFile.Free;
    end;
  finally
    Lock.EndRead;
  end;
end;

procedure THome.SessionsServiceEndSession(ASender: TObject;
  ASession: TAbstractWebSession; AReason: TEndSessionReason);
var
  IniFile: TIniFile;
begin
  //Save out the preferences here
```

**LISTING 23.4** Continued

```
  Lock.BeginWrite;
  if FCurrentUserName <> '' then
  begin
    try
      IniFile := TIniFile.Create(IniFileName);
      try
        IniFile.WriteString(FCurrentUserName, sFavoriteMovie,
        ➥ASession.Values[sFavoriteMovie]);
        IniFile.WriteString(FCurrentUserName, sPassWordHint,
        ➥ASession.Values[sPasswordHint]);
        IniFile.WriteString(FCurrentUserName, sLikesChocolate,
        ➥ASession.Values[sLikesChocolate]);
      finally
        IniFile.Free;
      end;
    finally
      Lock.EndWrite
    end;
  end;
end;
```

These event handlers store the data in an INI file, but there is no reason that you couldn't store the data in a database or any other persistent storage method.

The `Lock` variable is a global variable of type `TMultiReadExclusiveWriteSynchronizer`, and it is created in the Home page's initialization section. Because multiple sessions could be reading and writing to the INI file, this component makes reading and writing to the INI file thread-safe. Add the following declaration to the interface portion of your `wmHome` unit:

```
var
    Lock: TMultiReadExclusiveWriteSynchronizer;
```

And then add this to the initialization and finalization sections for the same unit:

```
Initialization
  ...
  Lock := TMultiReadExclusiveWriteSynchronizer.Create;
finalization
  Lock.Free;
```

This code also uses a function called `IniFileName` that is declared as follows:

```
const
      sIniFileName = 'DDG6Demo.ini';

...
```

```
function IniFileName: string;
begin
  Result := ExtractFilePath(GetModuleName(HInstance)) + sIniFileName;
end;
```

Add this to your `wmHome` unit, and you should have a fully functioning Web application that logs in users and tracks their preferences, even between sessions.

## Image Handling

Practically every Web application displays graphics. Graphics can enhance your application's appeal and functionality. Naturally, WebSnap makes including images and graphics in your applications as easy as, well, everything else WebSnap does. As you might expect, WebSnap will enable you to use graphics and images from any source you prefer—files, resources, database streams, and so on. If your image data can be put into a stream, it can be used in a WebSnap application.

Use the Internet toolbar to add another page to your application. Use a `TAdapterPageProducer`, publish the page, and require users to log in to gain access to it. Next name the page Images, and save the resulting unit as `wmImages`. After this is done, go to the Images Web module, add a `TAdapter` to the module, and give it the name ImageAdapter. Finally, double-click on ImageAdapter, and add two fields of type `TAdapterImageField`. Each of these will show a different way to display images.



First, you can display an image based on a URL. Highlight the first `AdaperImageField`, and set the `HREF` property to a fully qualified URL that points to an image on your system or anywhere on the Internet for that matter. For instance, if you want to look at the one-year history of Borland's stock price, set the `HREF` property to `http://chart.yahoo.com/c/1y/b/borl.gif`.

Double-click on the `TAdapterPageProducer` in the Images Web module, add an `AdapterForm`, and then to that add an `AdapterFieldGroup`. Set the adapter property of this new `AdapterFieldGroup` to the `ImageAdapter`. Then right-click again on the `AdapterFieldGroup` and select Add All Fields. Next, set the `ReadOnly` field of the `AdapterImageField` to `True`. If this property is `True`, it will display the image on your page. If it is set to `False`, it will give you an edit box and a button to look up a filename. Obviously, to see images, you should set this property to `True`. When you first look at the image, you will notice that the image has a pesky little caption. Most often you won't want that, so to get rid of it, set the `Caption` property to a single space. (Note that it won't accept a blank caption.) You should then see the chart appear in the Web Surface Designer as shown in Figure 23.12.


**FIGURE 23.12**
*The Web Surface Designer with a graphic in it from the ImageAdapterField.*

**NOTE**

If you want images referenced by relative links to show up at design time, you must add the directory where they reside to the Search Path on the Options page of the Web App Debugger.

Now you can display images based on a URL. At other times, however, you might want to get an image from a stream. The `AdapterImageField` component provides support for that as well. Select the second `AdapterImageField` from your `ImageAdapter` and open the Object Inspector. Go to the events page, and double-click on the `OnGetImage`. Put a JPG image in the same directory as application (the demo on the CD-ROM uses `athena.jpg`), and make your event handler resemble the following:

```
procedure TImages.AdapterImageField2GetImage(Sender: TObject;
  Params: TStrings; var MimeType: String; var Image: TStream;
  var Owned: Boolean);
begin
  MimeType := 'image\jpg';
  Image := TFileStream.Create('athena.jpg', fmOpenRead);
end;
```

This code is quite simple—Image is a stream variable that you create and fill with an image. Of course, the application needs to know what type of image it is getting, so you can return that information in the `MimeType` parameter. A `TFileStream` is a simple solution, but you could get the image from any source, such as a `BlobStream` from a database, or build the image on-the-fly and return it in a memory stream. Now when you run the application, you should see the JPG you chose right below the stock graphic.

## Displaying Data

Of course, you want your application to do more than the simple things it does so far. You'll certainly want to be able to display data from a database, both in tabular form and record by record. Naturally, WebSnap makes this easy, and you can build powerful database applications with only a modicum of code. By using the `TDatasetAdapter` and its built-in fields and actions, you can easily display data, as well as make additions, updates, and deletions to any database.

Actually displaying a dataset on a form is very easy. Add a new unit to your demo app—but this time make it a WebDataModule, using the third button on the Internet toolbar. This wizard is a simple one, so just accept the defaults. Then add a `TDatasetAdapter` from the WebSnap tab on the Component Palette, and a `TTable` from the BDE tab. Point the `TTable` to the DBDemos database, and then to the BioLife table. Then set the `Dataset` property of `DatasetAdapter1` to `Table1`. Finally, set `Table1.Active` to `True` to open the table. Name the Webdatamodule BioLife data, and save the unit as `wdmBioLife`.



**NOTE**

Your application is using a simple BDE-based Paradox table, but the `TDatasetAdapter` component will display data from any `TDataset` descendent. Note, too, that it isn't really a good idea to use a `TTable` in a Web application without explicit session support. The demo app does this just for ease of use, and to keep attention on the WebSnap features and not the data.

Then, for a change of pace, use the Object Treeview to set the properties of the components. If the Object Treeview isn't visible, select View, Object Treeview from the main menu. Select the DatasetAdapter, right-click on the Actions node, and select Add All Actions. Then, hook the `TTable` to the `TAdapterDataset` via its `Dataset` property. Select the Fields node and right-click, selecting Add All Fields. Do the same for the `TTable`, adding all the fields in the dataset to the WebDatamodule. Then, because WebSnap builds stateless servers for database operations, you must indicate a primary key for the dataset to enable client-requested navigation and data manipulation. WebSnap will do this all for you automatically after you specify the

primary key. Do this by selecting the Species_No Field in the Object Treeview and adding the `pfInKey` value to its `ProviderFlags` property.

Next, add a regular page to the application. Make it a Login Required page, give it a `TAdapterPageProducer`, and name the page Biolife. Save the unit as `wmBioLife`. Because you want to display the data in this new page, add the `wdmBioLife` unit name to the `uses` clause of your `wmBioLife` unit. Then, give the BioLife Web module the focus, and right-click on the `AdapterPageProducer` component. Right-click on the WebPageItems node just below it, select New Component, and select an `AdapterForm`. Select the `AdapterForm`, right-click it, and add an `AdapterErrorList`. Then add an `AdapterGrid`. Set the Adapter property of both components to the `DatasetAdapter`. Right-click on the `AdapterGrid` and select Add All Columns. Then select the Actions node under the `DatasetAdapter`, right-click it, and select Add All Actions. Next, select the Fields node, right-click, and add all the fields as well. You should now have all the properties properly set to display data.

Go to the BioLife Web module and double-click on the `AdaperPageProducer`. You should see the Web Surface Designer, with live data in it. If not, check to make sure that you have opened the table and hooked up all the Adapter properties for the components within the `DatasetAdapter`. The Notes field makes the table too long, so select the `AdapterGrid` in the upper left and the `ColNotes` component in the panel in the upper right, and then delete it. Now you should have something similar to that shown in Figure 23.13.

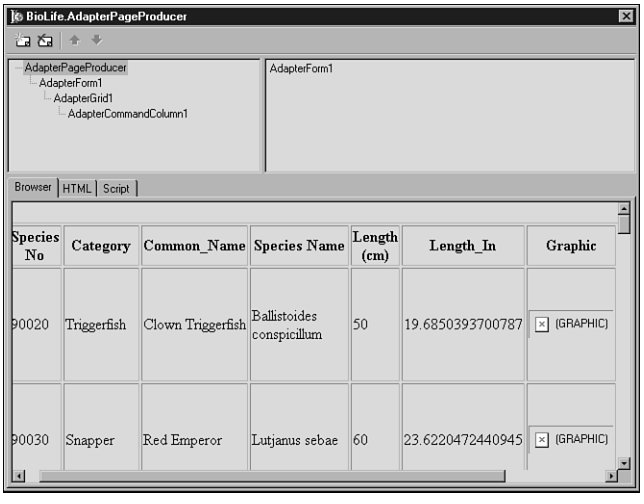


**FIGURE 23.13**

*The BioLife table in the Web Surface designer of a* `TAdapterPageProducer`*; the HTML table is produced by the* `TDatasetAdapter` *component.*

The graphics don't display at design time, but they will at runtime. Indeed, you can now compile and run the application, and you can view all the data on the BioLife page—all without writing a single line of code.

Of course, simply looking at the data isn't very useful. You'll likely want to manipulate individual records. Naturally, this is easy to do in WebSnap. Go to the Web Surface Designer and select the `AdapterGrid`. Right-click on it and add an `AdapterCommandColumn`. Then right-click on this and select the DeleteRow, EditRow, BrowseRow, and NewRow commands, as shown in Figure 23.14.

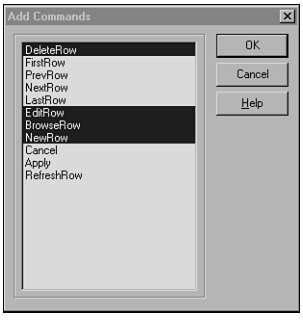


**FIGURE 23.14**
*Use the Add Commands dialog box to select the actions you want to take on individual rows of the dataset.*



Click OK. Then, vertically stack the buttons by setting the `DisplayColumns` property of the `AdapterCommandColumn` component to `1`. After you do that, you should see a collection of command buttons in the Web Designer (see Figure 23.15).

Currently, those buttons need to do something, and they'll need a page to display the individual record. Add another page to the project with a `TAdapterPageProducer` and require the user login to see the page. Name the page BioLifeEdit, and save the unit as `wmBioLifeEdit`. Add `wdmBioLife` to the `uses` clause so that you can access the data.

Double-click on the `TAdapterPageProducer` in the new Web module and add an `AdapterForm`. Then add an `AdapterErrorList`, an `AdapterFieldGroup`, and an `AdapterCommandGroup`. Right-click on the `AdapterFieldGroup` and add all the fields and then all the commands to the `AdapterCommandGroup`. The Web Surface Designer resembles what is shown in Figure 23.16.

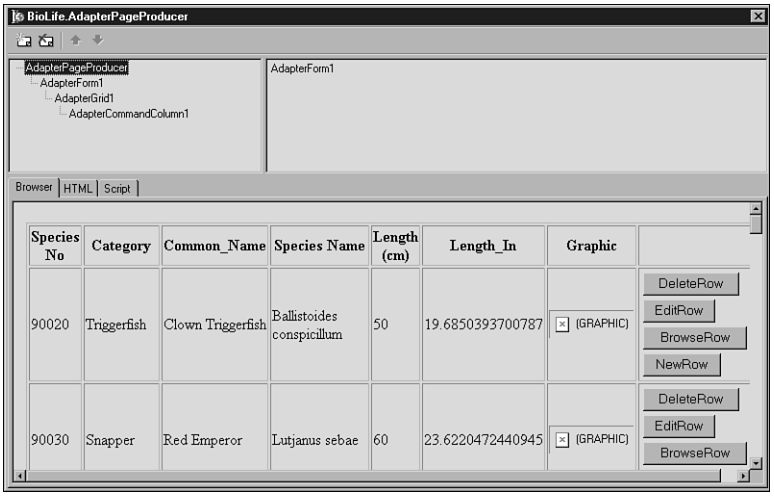


**FIGURE 23.15**
*The Demo application displaying the action buttons.*

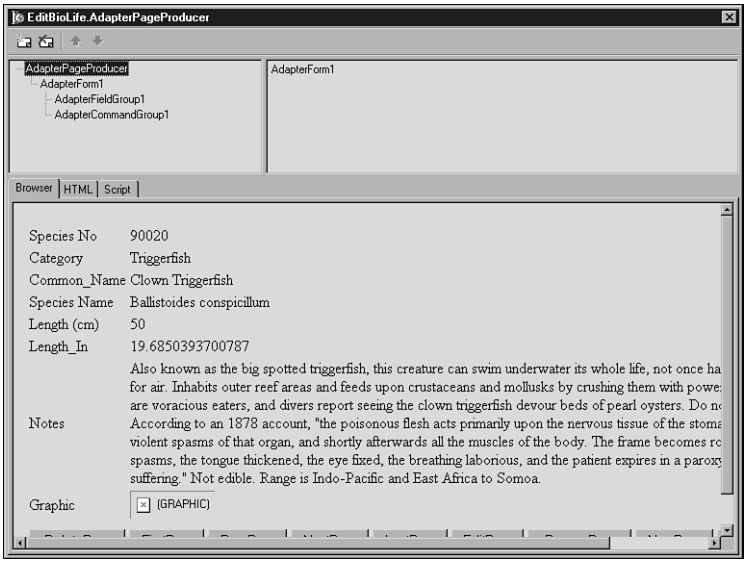


**FIGURE 23.16**
*The BioLifeEdit page with all the fields and actions added to the Web Surface Designer.*

Now, in order to use this page to edit a single record, go back to the `wmBioLife` unit where the grid is, and use the Object TreeView and the shift key to select all four buttons in the

AdapterCommandColumn. Set their page property to EditBioLife—the name of the page that will display a single record. Now, when you click the button in the grid, the EditBioLife page will be displayed. If you ask to browse the record, the data will be displayed as simple text. But if you ask to edit the record, the data will be displayed in edit boxes. The graphic field will even allow you to add a new graphic to the database by browsing for a new file. You can navigate through the dataset using the command buttons. And again, all this was accomplished without writing a single line of code—or script for that matter.

**NOTE**

You might want to tweak the presentation of the Notes field a little bit. By default, the TextArea control used when the page is in edit mode is quite small and doesn't wrap the text. You can select the FldNotes component and adjust the TextAreaWrap, DisplayRows, and DisplayWidth properties to get better results.

## Converting the Application to an ISAPI DLL

Your application has so far been run under the Web App Debugger, making it easy to debug and test. However, you certainly don't want to deploy the application that way. Instead, you'll likely want to make the application an ISAPI DLL so that it will always reside in memory and maintain all the session information needed to keep things in order.

Converting your application from a Web App Debugger based server to an ISAPI server is very straightforward. Simply create a new, blank ISAPI-based project, and remove all the units from it. Then, add in all units from the Web App version except the form. Then compile and run. It's that simple. In fact, you can maintain two projects that use the very same webmodules—one project for testing and another for deploying. Most of the demo applications in the WebSnap directory do this, and the demo application on the CD-ROM has both Web App Server and ISAPI projects. When deploying the new ISAPI DLL, be sure to include any HTML files that will need to be in the same directory as the DLL.

# Advanced Topics

So far, you have seen what can be considered the basics. You've created a WebSnap application that manages users, session information about those users, as well as manages and manipulates data. WebSnap does a lot more than that, however, and gives you more control over what your application can do. The next section covers some advanced topics that will allow you to more finely tune your WebSnap applications.

## LocateFileServices

The development of WebSnap Web applications usually requires the coordination of differing resources. HTML, server-side script, Delphi code, database access, and graphics all need to be properly tied together into a single application. Most often, many of these resources lie embedded in and are managed via an HTML file. WebSnap provides support for separating HTML from the implementation of a dynamic Web page, meaning that you can edit the HTML files separately from the Web application's binary file. However, by default, that HTML must reside in files in the same location as the binary file does. This isn't always convenient or possible, and there might be times when you want HTML to reside in locations away from your binary. Or it might be that you want to get HTML content from sources other than files, say a database.

WebSnap provides you the ability to get HTML from any source that you want. The `LocateFileService` component allows you to get HTML from any file location, include files, or any `TStream` descendant. Being able to access HTML from a `TStream` means that you can get the HTML from any source as long as it can be placed in a `TStream`.

For example, HTML can be streamed from a RES file embedded in your application's binary file. The demo application can show how this is done. Naturally, you'll need some HTML to embed. Using a text editor or your favorite HTML editor, take the wmLogin.html file as a template and save it in your demo application's directory as `embed.html`. Then, add some text to the file to note that the file is embedded in the RES file. That way, you'll know for sure that you have the right file when it is displayed.

Then, of course, you need to embed this HTML into your application. Delphi easily manages this via RC files, automatically compiling them and adding them to an application. Therefore, use Notepad or some text-handling tool to create a text file, and call it `HTML.RC`. Save it in the same directory as your demo application and add it to your project. Then, add this text to the RC file:

```
#define HTML 23  // HTML resource identifier
EMBEDDEDHTML HTML embed.html
```

When included in a Delphi project, Delphi will compile the RC file into a RES file and include it in your application.

When the HTML is in your app, create a new page with a `TPageProducer` and call it Embedded. Save the file as `wmEmbedded`. Then, go to the Home page and select the `LocateFileServices` component. Go to the Object Inspector Events page and double-click on the `OnFindStream` event. You'll get an event handler similar to this one:

```
procedure THome.LocateFileServiceFindStream(ASender: TObject;
  AComponent: TComponent; const AFileName: String;
  var AFoundStream: TStream; var AOwned, AHandled: Boolean);
```

```
begin

end;
```

The key parameters here are the `AFileName` and `AFoundStream` parameters. You'll use them to get the HTML from the embedded resources. Make your event handler resemble the following:

```
procedure THome.LocateFileServiceFindStream(ASender: TObject;
  AComponent: TComponent; const AFileName: String;
  var AFoundStream: TStream; var AOwned, AHandled: Boolean);
begin
  // we are hunting up the Embedded file
  if Pos('EMBEDDED', UpperCase(AFileName)) > 0 then   begin
    AFoundStream := TResourceStream.Create(hInstance, 'EMBEDDED', 'HTML');
    AHandled := True; // no need to look further
  end;
end;
```

`AFileName` will be the unqualified name of the HTML file that Delphi would use as a default. You can use that name to determine which resource to look up. `AFoundStream` will be `nil` when passed into event handler, so it is up to you to create a stream using the variable. In this case, `AFoundStream` becomes a `TResourceStream`, which grabs the HTML from the resources in the executable. Setting `AHandled` to `True` ensures that the LocateFileServices makes no further effort to find the HTML content.

Run the application, and you will see your HTML show up when you display the Embedded page.



## File Uploading

In the past, one of the more challenging tasks for a Web application developer is uploading files from the client to the server. It often involved dealing with the very arcane features of the HTTP specification and counting every byte passed very carefully. As you would expect, WebSnap makes this previously difficult task easy. WebSnap provides all the functionality for uploading a file inside a `TAdapter`, and your part isn't much more difficult than placing a file in a stream.

As usual, create another page in your application that will upload files to the server from the client. Name the page Upload and give it a `TAdapterPageProducer`. Then save the file as `wmUpload`. Then, drop a `TAdapter` on the form. Give the `TAdapter` a new `AdapterFileField`. This field will manage all the uploading of the files selected on the client. In addition, give the `Adapter` a single action and call it `UploadAction`.

Next, give the `AdapterPageProducer` an `AdapterForm` with an `AdapterErrorList`, an `AdapterFieldGroup`, and an `AdapterCommandGroup`. Connect the first two to `Adapter1`, and the

AdapterCommandGroup to the AdapterFieldGroup. Then add all the fields to the AdapterFieldGroup and all the actions to the AdapterCommandGroup. Change the caption on the button to Upload File. Figure 23.17 shows what you should see in the Surface Designer.

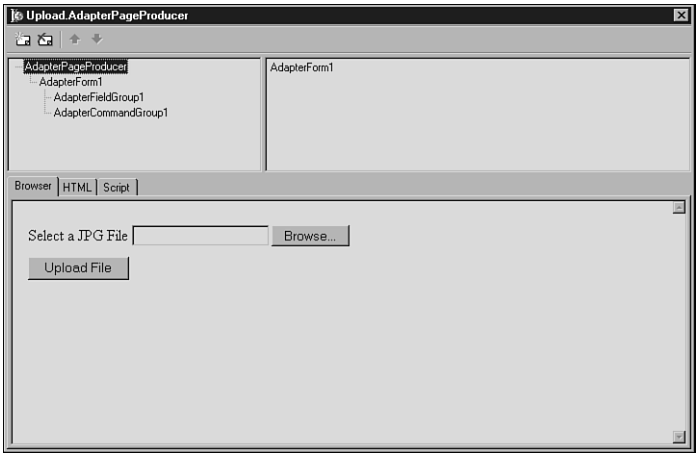


**FIGURE 23.17**
*The Web Surface Designer for the Upload page, with the Browse button automatically added.*

Code can be added in two places. The first place is to the Adapter1.AdapterFileField OnFileUpload event handler. The code there should resemble that in Listing 23.5.

**LISTING 23.5** OnFileUpload Event Handler

```
procedure TUpload.AdapterFileField1UploadFiles(Sender: TObject;
  Files: TUpdateFileList);
var
  i: integer;
  CurrentDir: string;
  Filename: string;
  FS: TFileStream;
begin
  // Upload file here
    if Files.Count <= 0 then
    begin
      raise Exception.Create('You have not selected any files to be uploaded');
    end;
    for i := 0 to Files.Count - 1 do
    begin
    // Make sure that the file is a .jpg or .jpeg
    if (CompareText(ExtractFileExt(Files.Files[I].FileName), '.jpg') <> 0)
```

**LISTING 23.5** Continued

```
        and (CompareText(ExtractFileExt(Files.Files[I].FileName), '.jpeg')
        ➥<> 0) then
    begin
      Adapter1.Errors.AddError('You must select a JPG or JPEG file to upload');
    end else
    begin
      CurrentDir := ExtractFilePath(GetModuleName(HInstance)) + 'JPEGFiles';
      ForceDirectories(CurrentDir);
      FileName := CurrentDir + '\' + ExtractFileName(Files.Files[I].FileName);
      FS := TFileStream.Create(Filename, fmCreate or fmShareDenyWrite);
      try
        FS.CopyFrom(Files.Files[I].Stream, 0); // 0 = copy all from start
      finally
        FS.Free;
      end;
    end;
  end;
end;
```



This code first checks to make sure that you have selected a file, and then it makes sure that you have selected a JPEG file. After it determines that you have done that, it takes the filename, ensures that the receiving directory exists, and puts the file into a `TFileStream`. The real work here is done behind the scenes by the `TUpdateFileList` class that manages all the HTTP esoterica and multi-part form handling needed to upload a file from the client to the server.

The second place to add code is in the `OnExecute` handler for the `UploadAction` in `Adapter1`. It is as follows:

```
procedure TUpload.UploadActionExecute(Sender: TObject; Params: TStrings);
begin
  Adapter1.UpdateRecords;
end;
```

which simply tells the `Adapter` to update its records and get the files that have been requested.

## Including Custom Templates

One thing you have likely noticed is that when you create a new page with the New Page Wizard, you only have two choices for the HTML in your application—the standard template or a blank template. The standard template is nice for things such as the demo application in this chapter, but when you start developing more sophisticated sites, you'll want to be able to automatically include your own HTML templates when adding pages to your applications. WebSnap allows you to do that.

You can add new templates to the selections in the New Page Wizard by creating and registering a descendent of `TProducerTemplatesList` in a design-time package. There is a demo package that does this in the `<Delphi>\Demos\WebSnap\Producer Template` directory. You can look at that package and add your own HTML/script templates to the RC file included in the package. Note that for this package to compile, you must first have compiled the package `<Delphi>\Demos\WebSnap\Util\TemplateRes.dpk`. After you compile and install these packages, you will have more templates to choose from in the New Page Wizard.

## Custom Components in `TAdapterPageProducer`

Much of the work of displaying HTML throughout this chapter has been done by `TAdapterPageProducer` components, and the components that are embedded within it. However, you certainly will want to customize the HTML therein beyond the standard code you have seen so far. WebSnap allows you to do this by creating your own components that plug in to the `TAdapterPageProducer`, allowing you to add your own custom HTML to the mix.

Your custom `TAdapterPageProducer` components must descend from `TWebContainedComponent` and implement the `IWebContent` interface. Because all the components must do this, it is a perfect opportunity to use an abstract class as in Listing 23.6.

**LISTING 23.6** Abstract Descendent Class of `TWebContainedComponent`

```
type

Tddg6BaseWebSnapComponent = class(TWebContainedComponent, IWebContent)
  protected
    { IWebContent }
    function Content(Options: TWebContentOptions; ParentLayout: TLayout):
    ➥string;
    function GetHTML: string; virtual; abstract;
  end;
```

This class is implemented like so:

```
function Tddg6BaseWebSnapComponent.Content(Options: TWebContentOptions;
  ParentLayout: TLayout): string;
var
  Intf: ILayoutWebContent;
begin
  if Supports(ParentLayout, ILayoutWebContent, Intf) then
    Result := Intf.LayoutField(GetHTML, nil)
  else
    Result := GetHTML;
end;
```

The abstract class implements the `Content` function only because the `GetHTML` function is declared as abstract. The `Content` function basically checks to see whether the containing component is a `LayoutGroup`. If it is `LayoutGroup`, the `Content` function places its content inside the `LayoutGroup`. Otherwise, `Content` simply returns the results of `GetHTML`. Descendent components, therefore, need only implement the `GetHTML` function, returning the appropriate HTML code, and they can be registered to work inside a `TAdapterPageProducer`.

The code on the CD-ROM implements two components that allow you to add HTML content to a `TAdapterPageProducer`, either as a string or as a file. The code for the `Tddg6HTMLCode` component is as shown in Listing 23.7.

**LISTING 23.7** `Tddg6HTMLCode` Component

```
  Tddg6HTMLCode = class(Tddg6BaseWebSnapComponent)
  private
    FHTML: TStrings;
    procedure SetHTML(const Value: TStrings);
  protected
    function GetHTML: string; override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property HTML: TStrings read FHTML write SetHTML;
  end;

constructor Tddg6HTMLCode.Create(AOwner: TComponent);
begin
  inherited;
  FHTML := TStringList.Create;
end;

destructor Tddg6HTMLCode.Destroy;
begin
  FHTML.Free;
  inherited;
end;

function Tddg6HTMLCode.GetHTML: string;
begin
  Result := FHTML.Text;
end;

procedure Tddg6HTMLCode.SetHTML(const Value: TStrings);
```

**LISTING 23.7** Continued

```
begin
  FHTML.Assign(Value);
end;
```

This is a pretty simple class. It merely provides a published property of type `TStrings` that will take any HTML code and then put it in the `TAdapterPageProducer` as is. The `GetHTML` function simply returns the HTML in string form. You can build components to return any HTML code you want to include—images, links, files, and other content. All descendent components have to do is to provide their HTML content in an overridden `GetHTML()` method. Note that there are supporting registration functions in the unit where the components are implemented. When creating components, be sure to register them in your unit similar to those on the CD-ROM. To use these components, merely install them in a design-time package, and the components will appear in the `TAdapterPageProducer`'s Web Surface Designer (see Figure 23.18).

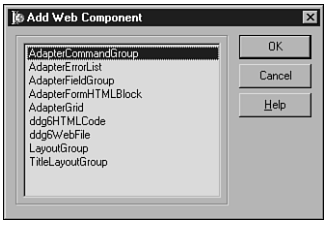


**FIGURE 23.18**
`TAdapterPageProducer` *components in the Web Surface Designer.*

# Summary

That's a quick overview of the power of WebSnap. This chapter barely scratched the surface of what WebSnap can do. Be sure to check out the numerous demo applications in the `<Delphi>\Demos\WebSnap` directory. Many of the demos add functionality to the standard slate of WebSnap components.

Clearly, WebSnap is a powerful technology, but it does take some effort to understand. However, once you get over the initial learning curve, you will soon be building powerful, database driven, dynamic Web sites with ease.