Chapter

29

Using WebSnap

WebSnap augments Web Broker with new components, wizards, and views—making it easier to build Web applications that contain complex, data-driven Web pages. WebSnap's support for multiple modules and for server-side scripting makes development and maintenance easier for teams of Delphi developers and Web designers.

WebSnap allows HTML design experts on your team to make a more effective contribution to Web server development and maintenance. The final product of the WebSnap development process includes a series of scriptable HTML page templates. These pages can be changed using HTML editors that support embedded script tags, like Microsoft FrontPage, or even a simple text editor. Changes can be made to the templates as needed, even after the application has been deployed. There is no need to modify the project source code at all, which saves valuable development time. Also, WebSnap's multiple module support can be used to partition your application into smaller pieces during the coding phases of your project. Your Delphi developers are now free to work more independently.

The dispatcher components automatically handle requests for page content, HTML form submissions, and requests for dynamic images. New components called adapters provide a means to define a scriptable interface to the business rules of your application. For example, the *TDataSetAdapter* object is used to make data-set components scriptable. You can use new producer components to quickly build complex, data-driven forms and tables, or to use XSL to generate a page. You can use the session component to keep track of end-users. You can use the user list component to provide access to user names, passwords, and access rights.

The Web application wizard allows you to quickly build an application that is customized with the components that you will need. The Web page module wizard allows you to create a module that defines a new page in your application. Or use the Web data module wizard to create a container for components that are shared across your Web application.

The page module views make it possible to see the result of server-side script without running the application. The Preview tab shows the page in an embedded browser.

The HTML Result tab shows the generated HTML. The HTML Script tab shows the page with server-side scripting, which is used to generate HTML for the page.

The following sections of this chapter explain how you use the WebSnap components to create a Web server application.

Fundamental WebSnap components

In order to build WebSnap Web server applications, you must first understand the fundamental components used in WebSnap development. They are:

- Web modules, which contain the components which make up the application and define pages
- Adapters, which provide an interface between HTML pages and the Web server application itself
- Page producers, which contain the routines which create the HTML pages to be served to the end user

Let us now examine each type of component in more detail. Afterwards, you will be ready to create your own WebSnap application.

Web modules

Web modules are the basic building block of WebSnap applications. Every WebSnap server application must have at least one Web module. More can be added as needed. There are four Web module types:

- *TWebAppPageModule*
- *TWebAppDataModule*
- *TWebPageModule*
- *TWebDataModule*

Web page modules (*TWebPageModule)* provide content to a page. Web data modules (*TWebDataModule*) act as a container for components shared across your application; they serve the same purpose in WebSnap applications that ordinary data modules serve in regular Delphi applications. You can include any number of Web page or data modules in your server application.

You may be wondering how many Web modules your application needs. Every WebSnap application needs one (and only one) Web application module of some type. Beyond that, you can add as many Web page or data modules as you need.

For Web page modules, a good rule of thumb is one per page style. If you intend to implement a page that can use the format of an existing page, you may not need a new Web page module. Modifications to an existing page module may suffice. If the page is very different from your existing modules, you will probably want to create a new module. For example, let's say you are trying to build a server to handle online catalog sales. Pages which describe available products might all share the same Web

page module, since the pages can all contain the same basic information types using the same layout. An order form, however, would probably require a different Web page module, since the format and function of an order form is different than that of an item description page.

The rules are different for Web data modules. Components which can be shared by many different Web modules should be placed in a Web data module to simplify shared access. You will also want to place components which can be used by many different Web applications in their own Web data module. That way you can easily share those items between applications. Of course, if neither of these circumstances apply you might choose not to use Web data modules at all. Use them the same way you would use regular data modules, and let your own judgment and experience be your guide.

Web application module types

Web application modules provide centralized control for business rules and non-visual components in the Web application. There are two types of Web application modules:

- Page Module: Selecting this type of module creates a content page. The page module contains a page producer which is responsible for generating the content of a page. The page producer displays its associated page when the HTTP request pathinfo matches the page name. The page can act as the default page when the pathinfo is blank.
- Data Module: Selecting this type of module does not create a content page. This module is used as a container for components shared by other modules—for example, database components used by two Web Page modules.

Web application modules act as a container for components that perform functions for the application as a whole—such as dispatching requests, managing sessions, and maintaining user lists. If you are already familiar with the Web Broker architecture, you can think of Web application modules as being similar to *TWebApplication* objects. Web application modules also contain the functionality of a regular Web module, either page or data, depending on the Web application module type. Your project can contain only one Web application module. You will never need more than one anyway; you can add regular Web modules to your server to provide whatever extra features you want.

Use the Web application module to contain the most basic features of your server application. If your server will maintain a home page of some sort, you may want to make your Web application module a *TWebAppPageModule* instead of a *TWebAppDataModule*, so you don’t have to create an extra Web page module for that page.

Web page modules

Each Web page module has a page producer associated with it. When a request is received, the page dispatcher analyses the request and calls the appropriate page module to process the request and return the content of the page.

Like Web data modules, Web page modules are containers for components. The difference between a Web data module and a Web page module is that a Web page module is used specifically to produce a Web page.

All web page modules have an editor view, called Preview, which allows you to preview the page as you are building it. You can take full advantage of the visual application development environment offered by Delphi.

Page producer component

Web page modules have a property that identifies the page producer component responsible for generating content for the page. (To learn more about page producers, see “Page producers” on page 29-9.) The WebSnap wizards automatically add a producer when creating a Web page module. You can change the page producer component later by dropping in a different producer from the WebSnap palette. However, if the page module has a template file, be sure that the content of this file is compatible with the producer component.

Page name

Web page modules have a page name that can be used to reference the page in an HTTP request or within the application's logic. A factory in the Web page module’s unit specifies the page name for the Web page module.

Producer template

Most page producers use a template. HTML templates typically contain some static HTML mixed in with transparent tags or server-side script. When page producers create their content, they replace the transparent tags with appropriate values and execute the server-side script to produce the HTML that is displayed by a client browser. (The XSLPageProducer is an exception to this. It uses XSL templates, which contain XSL rather than HTML. The XSL templates do not support transparent tags or server-side script.)

Web page modules may have an associated template file that is managed as part of the unit. A managed template file appears in the project manager and has the same base file name and location as the unit service file. If the Web page module does not have an associated template file then the properties of the page producer component specify the template.

Web data modules

Like standard data modules, Web data modules are a container for components from the palette. Data modules provide a design surface for adding, removing, and selecting components. The Web data module differs from a standard data module in the structure of the unit and the interfaces that the Web data module implements.

Use the Web data module as a container for components that are shared across your application. For example, you can put a dataset component in a data module and access the dataset from both:

- a page module that displays a grid, and
- a page module that displays an input form.

You can also use Web data modules to contain sets of components which can be used by several different Web server applications.

Structure of a Web data module unit

Standard data modules have a variable called the form variable, which is used to access the data module object. Web data modules replace this with a function. The purpose is the same. However, because WebSnap applications may be multi-threaded and may have multiple instances of a particular module that service multiple requests concurrently, this function is implemented to return the correct instance.

The unit also registers a factory. The factory specifies how the module should be managed by the WebSnap application. For example, flags indicate whether to cache the module and reuse it for other requests, or to destroy the module after a request has been serviced.

Adapters

Adapters define a script interface to your server application. They allow you to insert scripting languages into a page, and to retrieve information by making calls from your script code to the adapters. For example, you can use an adapter to define data fields to be displayed on an HTML page. A scripted HTML page can then contain HTML content and script statements that retrieve the values of those data fields. This is similar to the transparent tags used in Web Broker applications. Adapters also support actions which execute commands. For example, clicking on a hyperlink or submitting an HTML form can use adapter actions.

Adapters are useful because they simplify the task of creating HTML pages dynamically. If you use adapters in your application, you can take advantage of object-oriented script which supports conditional logic and looping. Without adapters and server-side script, you will need to write much more of your HTML generation logic in Pascal event handlers. Using adapters can significantly reduce development time.

See “Server-side scripting in WebSnap”on page 29-26 and “WebSnap server-side scripting reference” on page 29-30 for more details about scripting.

There are four types of adapter components that can be used to create page content: fields, actions, errors and records.

Fields

Fields are components that the page producer uses to retrieve data from your application and to display the content on a Web page. Fields can also be used to retrieve an image. In this case, the field returns the address of the image written to the Web page. When a page displays its content, a request is sent to the Web server application, which invokes the adapter dispatcher to retrieve the actual image from the field component.

Actions

Actions are components that execute commands on behalf of the adapter. When a page producer generates its page, the scripting language calls adapter action components to return the name of the action along with any parameters necessary to execute the command. For example, consider clicking a button on an HTML form to delete a row from a table. This returns, in the HTTP request, the action name associated with the button and a parameter indicating the row number. The adapter dispatcher locates the named action component and passes the row number as a parameter to the action.

Errors

Adapters keep a list of errors that occur while executing an action. Page producers can access this list of errors and display them in the Web page that the application returns to the end user.

Records

Some adapter components, such as *TDataSetAdapter*, represent multiple records. The adapter provides a scripting interface which allows iteration through the records. Some adapters support paging, and iterate over only the records on the current page.

Page producers

Use page producers to generate content on behalf of a Web page module. Producers provide the following functionality:

- They generate HTML content.
- They can reference an external file using the HTMLFile property, or the internal string using the HTMLDoc property.
- When the producers are used in conjunction with a Web page module, the template can be a file associated with a unit.
- Producers dynamically generate HTML which can be inserted into the template using transparent tags or active scripting. Transparent tags can be used in the same way as WebBroker applications. To learn more about using transparent tags, see “Converting HTML-transparent tags” on page 28-28. Active scripting support allows you to embed JScript or VBScript inside the HTML page.

First, let’s discuss the standard WebSnap method for using page producers. When you create a Web page module, you are asked to choose a page producer type in the Web Page Module wizard. You have many choices, but most WebSnap developers will want to prototype their pages by using an adapter page producer, *TAdapterPageProducer*. The adapter page producer lets you build a prototype Web page using a process analogous to the standard component model. You add a type of form, an adapter form, to the adapter page producer. As you need them, you can add adapter components (such as adapter grids) to the adapter form. Using adapter page

producers, you can create Web pages in a way similar to the standard Delphi technique for building user interfaces.

There are some circumstances where switching from an adapter page producer to a regular page producer is warranted. For example, part of the function of an adapter page producer is to dynamically generate script in a page template at runtime. You may decide that static script would be acceptable to help optimize your server. Also, users who are experienced with script may wish to make changes to the script directly. In this case, a regular page producer must be used to avoid conflicts between dynamic and static script. To learn how to change to a regular page producer, see "Advanced HTML design" on page 29-30

You can also use page producers in the same way as they are used in WebBroker applications, by associating the producer with a Web dispatcher action item. The advantages of using the Web page module are:

- the ability to preview the page's layout without running the application, and
- the ability to associate a page name with the module, so that the page dispatcher can call the page producer automatically.

Creating Web server applications with WebSnap

Hopefully you now have a good general understanding of WebSnap's architecture. If you look at the Delphi source code for WebSnap, you will discover that there are hundreds of objects in WebSnap. In fact, WebSnap is so rich in objects and features that you could spend a long time studying its architecture in detail before you understood it all. The thought of trying to comprehend WebSnap in every detail probably seems daunting. The question is, do you really need to understand the whole WebSnap system before you start developing your server application? Probably not.

You've probably dealt with similar problems before. Delphi as a whole is even more complex than WebSnap. If you're like most developers, you don't understand Delphi in every detail, nor do you feel a need to do so. Generally when you learn to use Delphi, you focus on learning the parts that you need to complete your current project. You learn other parts later, as you need them. You don't need to learn the ActionBands tools, which help you create a customizable graphic user interface, if you are developing a server application which has no GUI at all. You can just build your server application, learning the tools you need as you go, and worry about GUIs when you actually need to make one.

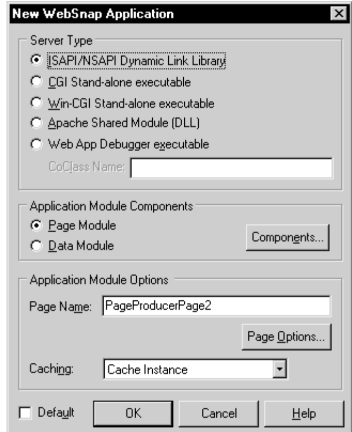
So, if you're like most developers, what's important to you right now is that you be able to use WebSnap. Fortunately, you're now ready to start working with WebSnap itself, so you can learn by doing. Here we will learn more about the basic WebSnap architecture by creating a new Web server application.

To create a new Web server application using the WebSnap architecture:

1. Select File | New | Other.

2 In the New Items dialog box, select the WebSnap tab and choose WebSnap Application.
3 A dialog box appears (as shown in Figure 29.1) which requires the following information:
    - Server type
    - Application module components
    - Application module options

Figure 29.1 The new WebSnap application dialog.



Server type

Select one of the following types of Web server application, depending on your application's type of Web server:

- ISAPI and NSAPI: Selecting this type of application sets up your project as a DLL with the exported methods expected by the Web server. It adds the library header to the project file, and adds the required entries to the uses list and to the exports clause of the project file.
- Apache: Selecting this type of application sets up your project as a DLL with the exported methods expected by the Apache Web server. It adds the library header to the project file and the required entries to the uses list and exports clause of the project file.
- CGI stand-alone: Selecting this type of application sets up your project as a console application, and adds the required entries to the uses clause of the project file.
- Win-CGI stand-alone: Selecting this type of application sets up your project as a Windows application and adds the required entries to the uses clause of the project file.

- Web Application Debugger executable: Selecting this type of application sets up an environment for developing and testing Web server applications. This type of application is not intended for deployment.

Choose the type of Web server application that communicates with the type of Web server your application will use.

In your development process, you might start by prototyping and debugging your Web server application as a Web Application Debugger executable. At some point you will want to convert your application to one of the other types of executable so it can be deployed. To convert your application, use the following steps:

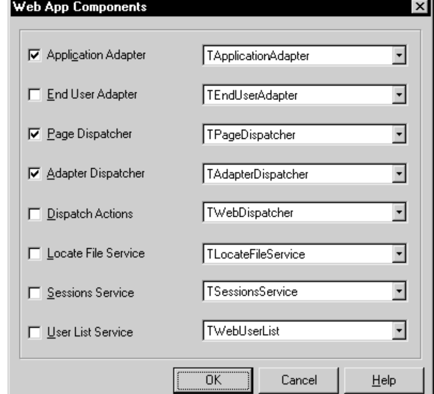
1. Open your Web Application Debugger executable project in the IDE.
2. Open the Project Manager using View | Project Manager. Expand your project so all its units are visible.
3. In the Project Manager, click the New button to create a new Web server application project. Double click the WebSnap Application item in the WebSnap tab. Select the appropriate options for your project, including the server type you want to use, then click OK.
4. Expand the new project in the Project Manager. Select any files appearing there and delete them.
5. One at a time, select each file in your Web Application Debugger project and drag it to the new project. When a dialog appears asking if you want to add that file to your new project, click Yes. Note that you should not drag the form unit to the new project. The form unit is used only by the Web Application Debugger executable.

Application module components

Application components provide the Web application's functionality. For example, including an adapter dispatcher component automatically handles HTML form submissions and the return of dynamically generated images. Including a page dispatcher automatically displays the content of a page when the HTTP request pathinfo contains the name of the page.

Selecting the Components button on the new WebSnap application dialog (see Figure 29.1) displays another dialog which allows you to select one or more of the Web application module components. The dialog, which is known as the Web App Components dialog, is shown in Figure 29.2.

Figure 29.2 The Web App Components dialog



Here is a brief explanation of the available components:

- Application Adapter: Contains information about the application, such as the title. The default type is *TApplicationAdapter*.
- End User Adapter: Contains information about the user, such as their name, access rights, and whether they are logged in. The default type is *TEndUserAdapter*. *TEndUserSessionAdapter* may also be selected.
- Page Dispatcher: Examines the HTTP request's pathinfo, and calls the appropriate page module to return the content of a page. The default type is *TPageDispatcher*.
- Adapter Dispatcher: Automatically handles HTML form submissions, and requests for dynamic images, by calling adapter action and field components. The default type is *TAdapterDispatcher*.
- Dispatch Actions: Allows you to define a collection of action items to handle requests based on pathinfo and method type. Action items call user-defined events, or request the content of page-producer components. The default type is *TWebDispatcher*.
- Locate File Service: Provides control over the loading of template files, and script include files, when the Web application is running. The default type is *TLocateFileService*.
- Sessions Service: Used to store information about an end-users that is needed for a short period of time. For example, you can use sessions to keep track of logged-in users, and to automatically log a user out after a period of inactivity. The default type is *TSessionService*.
- User List Service: Keeps track of authorized users, and their passwords and access rights. The default type is *TWebUserList*.

For each of the above components, the component types listed are the default types shipped with the Delphi software product. Users can create their own component types or use third-party component types.

Web application module options

If the selected application module type is page module, you can associate a name with the page by entering a name in the Page Name field in the dialog box. At runtime, the instance of this module can be either kept in cache, or removed from memory when the request has been serviced. Select either of the options from the Caching field. You can select more page module options through the Page Options button. You can set the following categories:

- Producer: The producer type for the page can be set to one of *AdapterPageProducer*, *DataSetPageProducer*, *InetXPageProducer*, *PageProducer*, or *XSLPageProducer*. If the selected page producer supports scripting, then use the Script Engine drop-down list to select the language used to script the page.

Note The *AdapterPageProducer* supports only JScript.

- HTML: When the selected producer uses an HTML template this group will be visible.
- XSL: When the selected producer uses an XSL template, such as *TXSLPageProducer*, this group will be visible.
- New File: Check New File if you want a template file to be created and managed as part of the unit. A managed template file will appear in the project manager and have the same file name and location as the unit source file. Uncheck New File if you want to use the properties of the producer component (typically the *HTMLDoc* or *HTMLFile* property).
- Template: When New File is checked, choose the default content for the template file from the Template drop-down. The "Default" template displays the title of the application, the title of the page, and hyperlinks to published pages.
- Page: Enter a page name and title for the page module. The page name is used to reference the page in an HTTP request or within the application's logic, whereas the title is the name that the end user will see when the page is displayed in a browser.
- Published: Check Published to allow the page to automatically respond to HTTP requests where the page name matches the pathinfo in the request message.
- Login Required: Check Login Required to require the user to log on before the page can be accessed.

You have now learned the basics of how to create a WebSnap server application. The WebSnap tutorial, which is the next section, walks you through the development process for a more complete application.

WebSnap tutorial

The following sections describe how to build a WebSnap application. Completing the tutorial will familiarize you with the WebSnap architecture and new concepts, by incorporating the new dispatcher and adapter components into a Web Page module.

The WebSnap application demonstrates how to use WebSnap HTML components to build an application that edits the content of a table.

Create a new application

Here you will learn how to create a new WebSnap application, which will eventually become the CountryTable application. CountryTable will display a table of information about various countries to users on the Web. Users can add and delete countries and edit information for existing countries. This simple example is meant to show you the fundamentals of WebSnap application development.

Step 1. Start the WebSnap application wizard

1 Run the Delphi application and select File | New | Other.
2 In the New Items dialog box, select the WebSnap tab and choose WebSnap Application.
3 In the New WebSnap Application dialog box:
   - Select the Web App Debugger Executable.
   - In the CoClass Name edit control, type **CountryTutorial.**
   - Select Page Module as the component type.
   - In the Page Name field type **Home.**
4 Click OK.

Step 2. Save the generated files and project

To save the Pascal unit file and project:

1 Select File | Save All.
2 In the File name field enter **HomeU.pas** and click Save.
3 In the File name field enter **CountryU.pas** and click Save.
4 In the File name field enter **CountryTutorial.dpr** and click Save.

Note Save the unit and the project to the same directory since the application will look for the HomeU.html file in the same directory as the executable.

Step 3. Specify the application title

The application title is the name displayed to the end user. To specify the application title:

1 Select View | Project Manager.
2 In the Project Manager window expand CountryTutorial.exe and double click the HomeU entry.
3 In the Object Inspector window (bottom left), select ApplicationAdapter from the pull down list.

4 In the Properties tab, enter **Country Tutorial** in the ApplicationTitle field.

5 Click on the Preview tab in the editor window. The application title is displayed at the top of the page, along with the page name, Home.

This page is extremely basic, of course. If you want to, you can improve the page by editing its HTML, either by using the HomeU.html editor tab or by using an external editor. For more information on how to edit the page template, see the "Advanced HTML design"section on page 29-24.

Create a CountryTable page

A Web page module is used to define a published page, and it also acts as a container for data components. Whenever a Web page needs to be returned to the end user, the Web page module will extract the necessary information from the data components it contains and use that information to help create a page. Here we will add a new module to our WebSnap application.

Step 1. Add a new Web page module

To add a new module:

1 Select File | New | Other.

2 In the New Items dialog box, select the WebSnap tab and choose WebSnap Page Module.

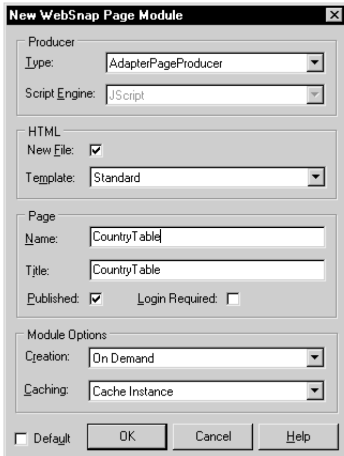
3 In the dialog box, set the Producer Type to AdapterPageProducer from the list.

4 In the Page Name field enter **CountryTable.**

5 Leave the rest of the fields and selection at their default values.

6 The dialog should appear as shown in Figure 29.3. Click OK.

Figure 29.3 The New WebSnap Page Module dialog for the CountryTable page.



The CountryTable module should now appear in the IDE as a window similar in appearance and function to a form. After saving the module, you will add new components to the CountryTable module.

Step 2. Save the new Web page module

Save the unit to the directory of the project file. When the application runs, it searches for the CountryTableU.html file in the same directory as the executable.

1 Select File | Save.

2 In the File name field, enter **CountryTableU.pas** and Click OK.

Add data components to the CountryTable module

A *TTable* component provides the data for the HTML table. The *TDataSetAdapter* component allows server side script to access the *TTable* component. Here we will add these data-aware components to our application.

If you are not familiar with Delphi database programming, you might not understand steps 1 and 2 below. You don't need to in order to complete this tutorial. WebSnap does not include new database functionality; it simply acts as an interface (through adapter components) to database components. To learn more about database programming, you can refer to the numerous database-related chapters of the Developer's Guide. For now, however, you can complete the tutorial and not worry about how the database works.

Step 1. Add data-aware components

1 Select View | Project Manager.

2 In the Project Manager window expand CountryTutorial.exe and double click the CountryTableU entry.
3 Select View|Object TreeView. The Object TreeView window (left side) becomes active.
4 Select the BDE tab in the component palette.
5 Select a Table component and add it to the CountryTable Web module.
6 Select a Session component and add it to the CountryTable Web module. The Session component is required because we are using a BDE component (*TTable*) in a multithreaded application.
7 Select the Session component, which is named Session1 by default, in the Web page module window or the Object TreeView window. This displays the Session component values in the Object Inspector window.
8 In the Object Inspector window, set the AutoSessionName property to *True*.
9 Select the Table component in the Web page module window or the Object TreeView window. This displays the Table component values in the Object Inspector window.
10 In the Object Inspector window, change the following properties:
   - Set the *DatabaseName* property to DBDEMOS .
   - In the *Name* property, type **Country**.
   - Set the *TableName* property to country.db.
   - Set the *Active* property to *True*.

Step 2. Specify a key field

The key field is used to identify records within a table. This becomes important when you add an edit page to the application. To specify a key field:

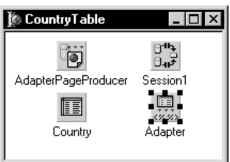
1 In the Object Tree View window, expand the Session and DBDemos node, and select the country.db node. This node is the Country Table component.
2 Right-click on the country.db node and select Fields Editor.
3 Right-click in the CountryTable.Country editor window and select the Add All Fields command.
4 Select the Name field from the list of added fields.
5 In the Object Inspector window, expand the *ProviderFlags* property.
6 Set the pfInKey property value to *True*.

Step 3. Add an adapter component

Now that we're finished adding database components, we return to WebSnap programming. To expose the data in the *TTable* through server-side scripting, you must include a data set adapter (*TDataSetAdapter*) component. To add such a component:

1 Select the WebSnap tab in the tool palette.
2 Select the DataSetAdapter component and add it to the CountryTable Web module.
3 In the Object Inspector window, change the following properties:
   - Set the DataSet field to Country.
   - In the Name field type **Adapter**.

Figure 29.4 The CountryTable Web page module



When you are finished, the CountryTable Web page module should look similar to what is shown in Figure 29.4. Since the elements in the module aren't visual, it doesn't matter where they appear in the module. What matters is that your module contains all the same components as those shown in the figure.

Create a grid to display the data

Step 1. Add a grid

To add a grid to display the data from the country table database:

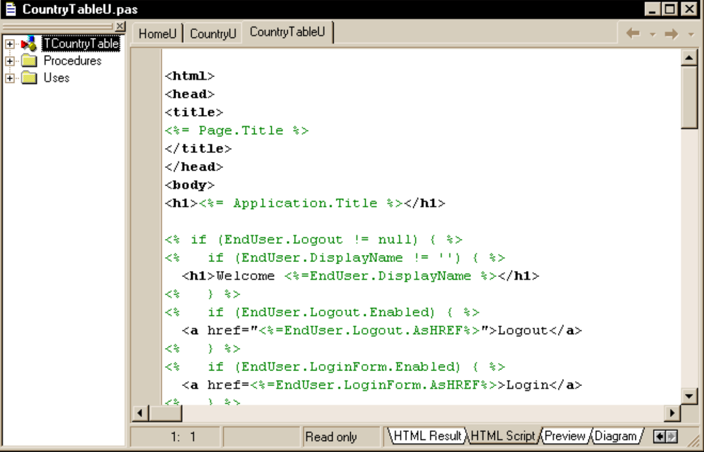
1 Select View | Project Manager.
2 In the Project Manager window, expand CountryTutorial.exe and double-click the CountryTableU entry.
3 Select View | Object TreeView. The Object TreeView window (left-hand side) becomes active.
4 Expand the *AdapterPageProducer* component. This component generates server-side script which will be used to quickly build an HTML table.
5 Right-click on WebPageItems entry and select New Component.
6 In the Add Web Component window, select AdapterForm, then click OK. An AdapterForm1 component appears in the Object TreeView window.
7 Right-click on AdapterForm1 and select New Component.
8 In the Add Web Component window, select AdapterGrid then click OK. An AdapterGrid1 component appears in the Object TreeView window.
9 In the Object Inspector window, set the Adapter property to Adapter.

Figure 29.5 The CountryTable Preview tab

To preview the page, select the CountryTableU.pas tab in the code editor window, and select the Preview tab at the bottom. If the Preview tab is not shown, use the right arrow at the bottom to scroll through the tabs. The preview should appear similar to Figure 29.5.

The Preview tab shows you what the final, static HTML page will look like in a Web browser. That page is generated from a dynamic HTML page which includes script. It is sometimes useful to see a text representation of the dynamic page with all the script commands shown. You can do so by selecting the HTML Script tab at the bottom of the editor window, which is shown in Figure 29.6.

Figure 29.6 The CountryTable HTML Script tab



The HTML Script tab shows a mixture of HTML and script. HTML and script are differentiated in the editor by font color and attributes. By default, HTML tags appear in boldfaced black text, while HTML attribute names appear in black and

HTML attribute values appear in blue. Script, which appears between the script brackets <% %>, is colored green. You can change the default font colors and attributes for these items in the Color tab of the Editor Properties dialog, which can be displayed by right-clicking on the editor and selecting Properties.

There are two other HTML-related editor tabs. The HTML Result tab shows the raw HTML contents of the preview. Note that HTML Result, HTML Script and Preview are all read-only. The last HTML-related editor tab, CountryTable.html, can be used for editing.

If you want to improve the look of this page, you can add HTML using either the CountryTable.html tab or an external editor at any time. For more information on how to edit the page template, see the "Advanced HTML design"section on page 29-24.

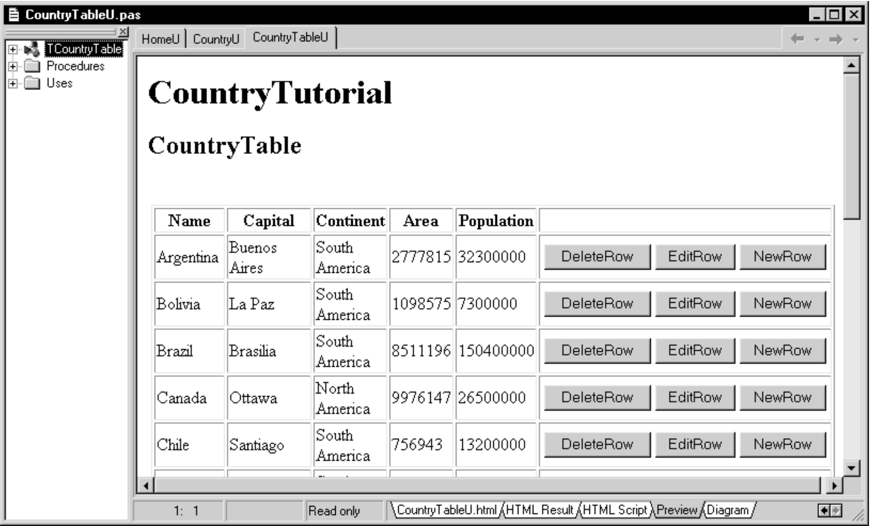
Step 2. Add editing commands to the grid

Users may need to update the content of the table by deleting, inserting or editing a row. To allow users to make such updates, add command components.

To add command components:

1. In the Object TreeView window for the CountryTable, expand the *AdapterPageProducer* component and all its branches.
2. Right-click on the AdapterGrid1 component and select Add All Columns.
3. Right-click on the AdapterGrid1 component and select New Component.
4. Select AdapterCommandColumn and then click OK. An AdapterCommandColumn1 entry is added to the AdapterGrid1 component.
5. Right-click on AdapterCommandColumn1 and choose Add Commands.
6. Multi-select the DeleteRow, EditRow, and NewRow commands; then click OK.
7. To preview the Page, click on the Preview tab at the bottom of the code editor. You will now see three new buttons (DeleteRow, EditRow and NewRow) at the end of each row in the table, as shown in Figure 29.7. When the application is running, pressing one of these buttons will cause the associated action to be performed.

Figure 29.7 CountryTable Preview after editing commands have been added.



Add an edit form

You will now create a Web page module to be the Edit form for the country table. Users will be able to change data in the CountryTable application through the Edit form. Specifically, when the user presses the EditRow or NewRow buttons, an Edit form will appear. When the user is finished with the Edit form, the modified information will appear in the table.

Step 1. Add a new Web page module

To add a new Web page module:

1. Select File | New | Other.
2. In the New Items dialog box, select the WebSnap tab and choose WebSnap Page Module.
3. In the dialog box, set the Producer Type to AdapterPageProducer from the list.
4. In the Page Name field, enter **CountryForm**.
5. Uncheck the Published box, so this page will not appear in a list of available pages on this site. The Edit form is accessed through the Edit button, and its contents depend on which row of the country table is to be modified.
6. Leave the rest of the fields and selections at their default values.
7. Click OK.

Step 2. Save the new module

Save the unit to the directory as the project file. When the application runs, it will look for the CountryFormU.html file in the same directory as the executable.

1. Select File | Save.
2. In the File name field enter **CountryFormU.pas** and click OK.

Step 3. Use the CountryTableU unit

Add CountryTableU unit to the **uses** clause to allow the module access to the Adapter component.

1. Select File | Use Unit.
2. Select CountryTableU from the list then click OK.
3. Select File | Save.

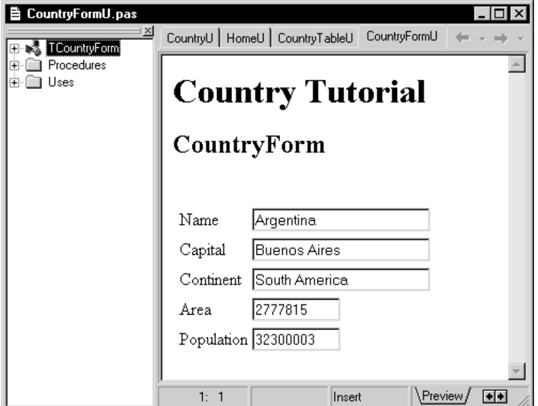
Step 4. Add input fields

Add components to the *AdapterPageProducer* component to generate data entry fields in the HTML form.

To add input fields:

1. Select View | Project Manager.
2. In the Project Manager window, expand CountryTutorial.exe and double-click the CountryFormU entry.
3. Select View | Object TreeView. The Object TreeView window (left-hand side) becomes active.
4. In the Object TreeView window, expand the *AdapterPageProducer* component, right-click on WebPageItems, and select New Component.
5. Select AdapterForm, then click OK. An AdapterForm1 entry appears in the Object TreeView window.
6. Right-click on AdapterForm1 and select New Component.
7. Select AdapterFieldGroup then click OK. An AdapterFieldGroup1 entry appears in the Object TreeView window.
8. In the Object Inspector window, set the Adapter property to CountryTable.Adapter. Set the AdapterMode property to Edit.
9. To preview the Page, click the Preview tab at the bottom of the code editor. Your preview should resemble the one shown in Figure 29.8.

Figure 29.8 The Preview tab for CountryForm



Step 5. Add buttons

Add components to the *AdapterPageProducer* component to generate the submit buttons in the HTML form. To add components:

1. In the Object TreeView, expand the *AdapterPageProducer* component and all its branches.
2. Right-click on AdapterForm1 entry and select New Component.
3. Select AdapterCommandGroup then click OK. An AdapterCommandGroup1 entry appears in the Object TreeView window.
4. In the Object Inspector window, set the DisplayComponent property to AdapterFieldGroup1.
5. Right-click on AdapterCommandGroup1 entry and select Add Commands.
6. Multi-select the Cancel, Apply, and Refresh Row commands, then click OK.
7. To preview the Page, click the Preview tab at the bottom of the code editor window. If the preview does not show the country form, click on the Code tab and then re-click the Preview tab. Your preview should resemble the one shown in Figure 29.9.

Figure 29.9 CountryForm with submit buttons.

Step 6. Link form actions to the grid page

When the user clicks a button, an adapter action is executed which carries out the described action. To specify which page to display after an adapter action is executed:

1. In the Object TreeView, expand AdapterCommandGroup1 to show the CmdCancel, CmdApply, and CmdRefreshRow entries.
2. Select CmdCancel. In the Object Inspector window, type **CountryTable** in the PageName property.
3. Select CmdApply. In the Object Inspector window, type **CountryTable** in the PageName property.

Step 7. Link grid actions to the form page

To specify which page to display after an adapter action is executed by pushing a button in the grid:

1. Select View | Project Manager.
2. In the Project Manager window, expand CountryTutorial.exe and double-click the CountryTableU entry.
3. In the Object TreeView window, expand the *AdapterPageProducer* component and all its branches, to show the CmdNewRow, CmdEditRow, and CmdDeleteRow entries. These entries appear under the AdapterCommandColumn1 entry.
4. Select CmdNewRow. In the Object Inspector window, type **CountryForm** in the PageName property.
5. Select CmdEditRow. In the Object Inspector window, type **CountryForm** in the PageName property.

6 To verify that the application is working and that all buttons perform some action, run the application. When you run the application, you are running a server. To check that the application is working, you must view it in a Web Browser. You can do this by launching it from the Web Application debugger. For information on how to do this, see "Advanced HTML design" on page 29-30.

Note There will be no indication of database errors, such as an invalid type. For example, try adding a new country with an invalid value (for example, 'abc') in the Area field.

Run the completed application

To run the completed application:

1. Select Run | Run. A form is displayed. Web App Debugger executable Web applications are COM servers, and the form you see is the console window for the COM server. The first time you run the project, it registers the COM object that the Web App Debugger can access directly.
2. Select Tools | Web App Debugger.
3. Click on the default URL link to display the ServerInfo page. The ServerInfo page displays the names of all registered Web Application Debugger executables.
4. Select CountryTutorial in the drop-down list and click on the Go button.

Add error reporting

To report errors to the end user, an AdapterErrorList component is used to display errors that occur while executing adapter actions that edit the country table.

Step 1. Add error support to the grid

1. In the Object TreeView for CountryTable, expand the *AdapterPageProducer* component and all its branches to show AdapterForm1.
2. Right-click on AdapterForm1 and select New Component.
3. Select AdapterErrorList from the list, then click OK. An AdapterErrorList1 entry appears in the Object TreeView window.
4. Move AdapterErrorList1 above AdapterGrid1 (either by dragging it or by using the upward-pointing arrow in the Object TreeView toolbar).
5. In the Object Inspector window, set the Adapter property to Adapter.

Step 2. Add error support to the form

1. In the Object TreeView for CountryForm, expand the *AdapterPageProducer* component and all its branches to show AdapterForm1.
2. Right-click on AdapterForm1 and select New Component.

3. Select AdapterErrorList from the list, then click OK. An AdapterErrorList1 entry appears in the Object TreeView window.
4. Move AdapterErrorList1 above AdapterFieldGroup1 (either by dragging it or by using the upward-pointing arrow in the Object TreeView toolbar).
5. In the Object Inspector window, set the Adapter property to CountryTable.Adapter.

Step 3. Test the error-reporting mechanism

To observe the error-reporting mechanism we will see here, you must first make a small change to the Delphi IDE. Select Tools | Debbugger Options. In the Language Exceptions tab, make sure the Stop on Delphi Exceptions checkbox is unchecked, which will allow the application to proceed when exceptions are detected. Now, to test for grid errors:

1. Run the application, and browse to the CountryTable page using the Web Application debugger. For information on how to do this, see "Advanced HTML design" on page 29-30.
2. Start up another instance of your browser and browse to the CountryTable page.
3. Click the DeleteRow button on the first row in the grid.
4. Without refreshing the second browser, click the DeleteRow button on the first row in the grid.
5. An error message, "Row not found in Country," will be displayed above the grid.

To test for form errors:

1. Run the application, and browse to the CountryTable page using the Web Application debugger.
2. Click on the EditRow Button.
3. The CountryForm page is displayed.
4. Change the area field to 'abc', and click the Apply Button.
5. An error message ("Invalid value for field 'Area'") will be displayed above the first field.

You have now completed the WebSnap tutorial. You might want to recheck the Stop on Delphi Exceptions checkbox before continuing.

Advanced HTML design

Using adapters and adapter page producers, WebSnap makes it easy to create scripted HTML pages in your Web server application. You can create a Web front end for your application data using WebSnap tools which may suit all of your needs. One powerful feature of WebSnap, however, is the ability to incorporate Web design expertise from other sources into your application. In this section, we will discuss

some strategies for expanding the Web server design and maintenance process to include other tools and non-programmer team members.

The end products of WebSnap development are your server application and HTML templates for the pages that the server produces. The templates include a mixture of scripting and HTML. Once they have been generated initially, they can be edited at any time using any HTML tool you like. (It would be best to use a tool that supports embedded script tags, like Microsoft FrontPage, to ensure that the editor doesn't accidentally damage the script.) The ability to edit template pages outside of the IDE can be used many ways.

For example, Delphi developers can edit the HTML templates at design time using any external editor they prefer. This allows them to use advanced formatting and scripting features that may be present in an external HTML editor but not in Delphi. To enable an external HTML editor from the IDE, use the following steps:

1. From the main menu, select Tools | Environment Options. In the Environment Options dialog, click on the Internet tab.
2. In the Internet File Types box, select HTML and click the Edit button to make the Edit Type dialog visible.
3. In the Edit Action box, select an action associated with your HTML editor. For example, if your HTML editor is the default HTML editor on your system, simply select Edit from the dropdown menu. Click OK twice to close the Edit Type and Environment Options dialogs.
4. To edit an HTML template, open the unit which contains that template. In the Edit window, right-click and select html Editor from the popup menu. The template will now appear in an external editor window for the external HTML editor you selected.

After the product has been deployed, you may wish to change the look of the final HTML pages. Perhaps your software development team is not even responsible for the final page layout. That duty may belong to a dedicated Web page designer in your organization, for example. Your page designers may not have any experience with Delphi development. Fortunately, they don't have to. They can edit the page templates at any point in the product development and maintenance cycle, without ever manipulating the source code. Here is an example of how that can be done.

In the development process, the Delphi development team creates the Web server application with prototype page templates produced by adapter page producers or page producers. After the software development team is finished, they deliver the prototype template pages to an HTML professional who puts them into their final format. They can add content or client-side scripting (using a language like JavaScript) to the pages, or perform any other editing tasks necessary. When the final HTML editing is complete, the templates can be deployed to the Web server hosting the server application. At any time thereafter, the pages can be altered as needed by the HTML designer, without ever modifying the project source code.

Obviously, this is not the only process for server development, but it does demonstrate how WebSnap HTML templates can make server development and maintenance more efficient.

Manipulating server-side script in HTML files

HTML in page templates can be modified at any time in the development cycle. Server-side scripting can be a different matter, however. It is always possible to manipulate the server-side script in the templates outside of Delphi, but it is not recommended for pages generated by an adapter page producer. The adapter page producer is different from ordinary page producers in that it can change the server-side scripting in the page templates at runtime. It can be difficult to predict how your script will act if other script is added dynamically. If you want to manipulate script directly, make sure that your Web page module contains a page producer instead of an adapter page producer.

If you have a Web page module which uses an adapter page producer, you can convert it to use a regular page producer instead by using the following steps:

1. In the module you wish to convert (let's call it ModuleName), copy all of the information from the HTML Script tab to the ModuleName.html tab, replacing all of the information that it contained previously.
2. Drop a page producer (which can be found on the Internet tab of the component palette) on your Web page module.
3. Set the page producer's ScriptEngine property to match that of the adapter page producer it is replacing.
4. Change the page producer in the Web page module from the adapter page producer to the new page producer. Click on the Preview tab to verify that the page contents are what they were before.
5. The adapter page producer has now been bypassed. You may now delete it from the Web page module.

Server-side scripting in WebSnap

Page producer templates can include scripting languages such as JScript or VBScript. The page producer executes the script in response to a request for the producer's content. Because the Web server application evaluates the script, it is called server-side script, as opposed to client-side script (which is evaluated by the browser).

This section is meant to give you a conceptual overview of server-side scripting and how it is used by WebSnap applications. The next section, the "WebSnap server-side scripting reference", has much more detailed information about script objects and their properties and methods. You can think of it as an API reference for server-side scripting, similar to the object descriptions for Delphi found in the help files. The next section also contains detailed script examples which show you exactly how script can be used to generate HTML pages.

Although server-side scripting is a valuable part of WebSnap, it is not essential that you use scripting in your WebSnap applications. Scripting is used for HTML generation and nothing else. It allows you to insert application data into an HTML page. In fact, almost all of the properties exposed by adapters and other script-

enabled objects are read-only. Server-side script isn't used to change application data, which is still managed by components and event handlers written in Pascal .

There are other ways to insert application data into an HTML page. You can use Web Broker's transparent tags, or some other tag-based solution, if you prefer. For example, there are several projects installed in the WebSnap Demos folder which use XML and XSL instead of scripting. Without scripting, however, you will be forced to write most of your HTML generation logic in Pascal, which will increase your development time.

The scripting used in WebSnap is object-oriented and supports conditional logic and looping, which can greatly simplify your page generation tasks. For example, your pages may include a data field which can be edited by some users but not others. With scripting, conditional logic can be placed in your template pages which displays an edit box for authorized users and simple text for others. With a tag-based approach, you must program such decision-making into your HTML generating source code.

Active scripting

WebSnap relies on *active scripting* to implement server-side script. Active scripting is a technology created by Microsoft to allow a scripting language to be used with application objects through COM interfaces. Microsoft ships two active scripting languages, VBScript and JScript. Support for other languages is available through third parties.

Script engine

The page producer's *ScriptEngine* property identifies the active scripting engine that evaluates the script within a template.

Script blocks

Script blocks are delimited by <% and %>. The script engine evaluates any text inside script blocks. The result becomes part of the page producer's content. The page producer writes text outside of a script block after translating any embedded transparent tags. Script blocks can also enclose text, allowing conditional logic and loops to dictate the output of text. For example, the following JScript block generates a list of five numbered lines:

```
<ul>
<% for (i=0;i<5;i++) { %>
    <li>Item <% Response.Write(i) %></li>
<% } %>
</ul>
```

The following script block is equivalent:

```
<ul>
<% for (i=0;i<5;i++) { %>
```

```
    <li>Item <%=i %></li>
<% } %>
</ul>
```

The <%= delimiter is short for *Response.Write*.

Creating script

Developers can take advantage of WebSnap features to automatically generate script.

Wizard templates

When they create a new WebSnap application or page module, WebSnap wizards provide a template field that is used to select the initial content for the page module template. For example, the template called "Default" generates JScript to display the application title, page name, and links to published pages.

TAdapterPageProducer

The *TAdapterPageProducer* builds forms and tables by generating HTML and JScript. The generated JScript calls adapter objects to retrieve field values, field image parameters, and action parameters.

Editing and viewing script

The WebSnap surface designer provides a view of your Web Page modules which lets you preview a scripted page. Use the HTML Result tab to view the HTML resulting from the executed script. Use the Preview tab to view the result in a browser. The HTML Script tab is available when the Web Page module uses *TAdapterPageProducer*. The HTML Script tab displays the HTML and JScript generated by the *TAdapterPageProducer* object. Consult this view to see how to write script that builds HTML forms to display adapter fields and execute adapter actions.

Including script in a page

A template can include script from a file or from another page. To include script from a file, use the following code statement:

```
<!-- #include file="filename.html" -->
```

When the template includes script from another page, the script is evaluated by the including page, use the following code statement to include the unevaluated content of page1.

```
<!-- #include page="page1" -- >
```

Script objects

Script objects are either VCL or CLX objects that can be referenced by script. You make VCL or CLX objects available for scripting by registering an *IDispatch* interface to the object with the active scripting engine. The following objects are available for scripting:

- **Application**—The application object (which may be null) provides access to the application adapter of the Web Application module. The following JScript block writes the application title:

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

- **EndUser**—The EndUser object provides access to the end-user adapter of the Web Application module. The following JScript block writes the end-user name:

  ```
  <%= EndUser.DisplayName %>
  ```

- **Session**—The session object provides access to the session object of the Web Application module. The following JScript block writes the session ID:

  ```
  <%= Session.SessionID %>
  ```

- **Pages**—The pages object (*Pages*) provides access to the application pages. The following JScript block writes links to all published pages:

  ```
  <%  e = new Enumerator(Pages)
       for (; !e.atEnd(); e.moveNext())
       {
         if (e.item().Published)
         {
             Response.Write('<A HREF="' + e.item().HREF +'">' + e.item().Title + '</A>')
         }
       }
  %>
  ```

  Note that the editor's Preview tab will not display the proper result of this script block. The pages object is always empty at design time because the Web page module factories have not been registered.

- **Modules**—The modules object provides access to the application modules. The following JScript block writes the content of an adapter field in a module called DM.

  ```
  <%= Modules.DM.Adapter1.Field1.DisplayText %>
  ```

- **Page**—The Page object provides access to the current page. The following JScript block writes the title of the current page:

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

- **Producer**—The Producer object provides access to the page producer of the Web Page module. The following JScript block evaluates a transparent tag before writing the content:

  ```
  <% Producer.Write('Here is a tag <#TAG>') %>
  ```

Note that the editor's Preview tab will probably not display the proper result of this script block. The event handlers that usually replace transparent tags do not execute unless the application is running.

- **Response—**The Response object provides access to the WebResponse. Use this object when tag replacement is not desired.

```
<% Response.Write('Hello World!') %>
```

- **Request—**The Request object provides access to the WebRequest. The following JScript block displays the pathinfo.

```
<%= Request.PathInfo %>
```

- **Adapter Objects—**All of the adapter's components on the current page can be referenced without qualification. Adapter's in other modules must be qualified using the Modules objects. The following JScript block displays the text value of the *FirstName* field from of all rows of Adapter1:

```
<% e = new Enumerator(Adapter1.Records) %>
  <% for (; !e.atEnd(); e.moveNext()) %>
  <% { %>
      <p><%= Adapter1.FirstName.DisplayText %>
  <% } %>
```

For more complete descriptions of these objects, see the next section, the “WebSnap server-side scripting reference”.

WebSnap server-side scripting reference

By this point you should have a good understanding of what scripting can do in your WebSnap server application. There are some developers who want to understand the details of how script is used in HTML template pages. This section is meant for those users. Here we will examine the mechanics of how script is used by WebSnap to generate dynamic HTML pages.

This section should be of particular interest to those developers who use page producers instead of adapter page producers in their Web page modules. The information contained here will help you do your own script programming in your page templates. This section should also be of interest to adapter page producer users who simply want to better understand the output of the adapter page producer.

Global objects

The following global objects can be referenced with server-side script.

Table 29.1 WebSnap global objects

Object	Description
Application	Use the Application object to access fields and actions of the application adapter, such as the Title field.
EndUser	Use to EndUser object to access fields and actions of the end-user adapter, such as the DisplayName for the end-user, the Login action, and Logout action.
Modules	Use the Modules object to reference a data module or page module by name. The Modules variables can also be used to enumerate the modules of the application.
Page	Use the Page object to access the properties of the page being generated such as the page Title.
Pages	Use the Pages object to reference a registered page by name. The Pages variables can also be used to enumerate the registered pages of the application.
Producer	Use the Producer object to write HTML content that may include transparent tags.
Request	Use the Request object to access the properties and methods of the HTTP request.
Response	Use the Response object to write HTML content to the HTTP response.
Session	Use the Session object to access the properties of the end-user's session.

Application

See also AdapterType

The Application object provides access to information about the application.

Use the Application object to access fields and actions of the application adapter, such as the Title field. The Application object is an Adapter so it can be customized with additional fields and actions. Fields and actions that have been added to the application adapter can be accessed by name as properties of the Application object.

Properties

Designing: Boolean, read

See also Example 1

Indicates whether the web application is being designed in the IDE.

Use the Designing flag to conditionally generate HTML that must be different when in design mode than when the Web application is running.

ModulePath: text, read

See also QualifyFileName

Identifies the location of the web application executable.

Use the ModulePath to construct file names that are in the same directory as the executable.

ModuleFileName: text, read

See also QualifyFileName

Identifies the fully qualified file name of the executable.

Title: text, read

See also Example 18

Provides the title of the application.

The Title property has the value of TApplicationAdapter.Title component property. Typically this value is displayed at the top of HTML pages.

Methods

QualifyFileName(FileName): text

See also Example 1

Make a relative filename or directory reference an absolute reference.

QualifyFileName uses the directory location of the web application executable to qualify a filename that is not fully qualified. A fully qualified filename is returned. If the filename parameter is fully qualified, the filename is returned without change. If in design mode, the filename parameter is qualified with the directory location of the project file.

EndUser

See also AdapterType

The EndUser object provides access to information about the current end-user.

Use to EndUser object to access fields and actions of the end-user adapter, such as the DisplayName for the end-user, the Login action, and Logout action. Fields and actions that have been added to the enduser adapter can be accessed by name as properties of the EndUser object.

Properties

DisplayName: text, read

See also Example 19

Provides the name of the end-user.

LoggedIn: Boolean, read

See also Example 19

Indicates whether the end-user is logged in.

LogInFormAction: AdapterAction, read

See also Example 19, AdapterActionType

Provides the adapter action used to login a user.

LogoutAction: AdapterAction, read

See also Example 19, AdapterActionType

Provides the adapter action used to logout a user.

Modules

See also Example 2, Example 20

Modules provides access to all modules that have been instantiated or activated to service the current HTTP request.

To references a particular module use the module's name as a property of the Modules variable. To enumerate all modules within the application, create an enumerator using the Modules object.

Page

See also Example 5, PageType

Page provides access to the properties of the page being generated.

See PageType for a description of the properties and methods of the Page object.

Pages

See also Example 5

Pages provides access to all pages registered by the application.

To references a particular page use the page's name as a property of the Pages variable. To enumerate all pages within the application, create an enumerator using the Pages object.

Producer

See also Response

Use the Producer object to write text containing transparent tags. The tags will be translated by the page producer and then written to the HTTP response. If the text does not contain transparent tags, use the Response object for better performance.

Properties

Content: text, read/write

Provides access to the content portion of the HTTP response.

Use the Content to read or write the entire content portion of the HTTP response. Setting Content translates transparent tags. If not using transparent tags, set Response.Content for better performance.

Methods

Write(Value)

Appends to the content portion of the HTTP request with support for transparent tags.

Use the Write method to append to the content portion of the HTTP request's content. Write translates transparent tags (e.g. Write('Translate this: <#MyTag>')). If you are not using transparent tags, use Response.Write for better performance.

Request

Provides access to the HTTP request.

Use properties of the Response object to access information about the HTTP request.

Properties

Host: text, read

Reports the value of the Host header of the HTTP request.

Host is the same as TWebRequest.Host.

PathInfo: text, read

Contains the PathInfo portion of the URL.

PathInfo is the same as the TWebRequest.InternalPathInfo property.

ScriptName: text, read

Contains the script name portion of the URL, which specifies the name of a Web server application.

ScriptName is the same as the TWebRequest.InternalScriptName property.

Response

See also Producer

Provides access to the HTTP response. Use the Response object to write to the content portion of the HTTP response. If writing transparent tags, use the Producer object instead of the Response object.

Properties

Content: text, read/write

Provides access to the content portion of the HTTP response.

Use Content to read or write the entire content portion of the HTTP response.

Methods

Write(Value)

See also Example 5

Appends to the content portion of the HTTP request.

Use the Write method to append to the content of the HTTP request's content. Write does not translate transparent tags.

Use Producer.Write to write a string containing one or more transparent tags.

Session

The Session object provides access to the session ID and values.

A session is used to keep track of information about the end-user for a short period of time.

Properties

SessionID.Value: text, read/write

Provides access to the id of the current end-user's session.

Values(Name): variant, read

Provides access to values stored in current end-user's session.

Object types

Some objects have properties that are objects. The following table lists the object types. Note that these type names are for documentation purposes. They are simply types of objects, not references to any specific instances in a WebSnap application, so server-side script does not recognize these names.

Table 29.2 WebSnap object types

Type name	Description
AdapterType	AdapterType defines the properties and methods of an adapter. Adapters can be accessed by name as a property of a Module.
AdapterActionType	AdapterActionType defines the properties and methods of an adapter action. Actions are referenced by name as a property of an adapter.
AdapterErrorsType	AdapterErrorsType defines the Errors property of an adapter. The Errors property is used to list errors that occurred when executing an action or generating a page.
AdapterFieldType	AdapterFieldType defines the properties and methods of an adapter field. Fields are referenced by name as a property of an Adapter.

Table 29.2 WebSnap object types

Type name	Description
AdapterFieldValuesType	AdapterFieldValuesType defines the properties and methods of an adapter field's Values property.
AdapterFieldValuesListType	AdapterFieldValuesListType defines the property and methods of an adapter field's ValuesList property.
AdapterHiddenFieldsType	AdapterHiddenFieldsType defines the HiddenFields and HiddenRecordFields property of an adapter.
AdapterImageType	AdapterImageType defines the Image property of adapter fields and adapter actions.
ModuleType	ModuleType defines the properties of a Module. A Module can be accessed by name as a property of the Modules variable.
PageType	PageType defines the properties of a page. A page can be accessed by name as a property of the Pages object. The page being generated can be accessed using the Page object.

AdapterType

Defines the properties and methods of an adapter. Adapters can be accessed by name as a property of a Module.

Adapters contain field components and action components that represent data items and commands, respectively. Server-side script statements access the value of adapter fields and the parameters of adapter actions in order to build HTML forms and tables.

Properties

Actions: Enumerator

See also Fields, Example 8

Enumerates the action objects. Use the Actions property to loop through the actions of an adapter.

CanModify: Boolean, read

See also CanView, AdapterFieldType.CanView

Indicates whether the end-user has rights to modify fields of this adapter. Use the CanModify property to dynamically generate HTML that is sensitive to the end-user's rights. For example, use an <input> element if CanModify is True. Use <p> if CanModify is False.

CanView: Boolean, read

See also CanModify, AdapterFieldType.CanModify

Indicates whether the end-user has rights to view fields of this adapter. Use the CanModify property to dynamically generate HTML that is sensitive to the end-user's rights.

ClassName_: text, read

See also Name_

Identifies the VCL class name of the adapter component.

Errors: AdapterErrors, read

See also AdapterErrorsType, Example 7

Enumerates the errors that were detected while processing an HTTP request. Adapters capture errors that occur will generating an HTML page or executing an adapter action. The Errors object is used to enumerate the errors and display error messages on an HTML page.

Fields: Enumerator

See also Actions

Enumerates the field objects. Use the Fields property to loop through the fields of an adapter.

HiddenFields: AdapterHiddenFields

See also HiddenRecordFields, AdapterHiddenFieldsType, Example 10, Example 22

HiddenFields defines the hidden input fields that pass adapter state information. An example of state information is TDataSetAdapter's mode. "Edit" and "Insert" are two possible mode values. When TDataSetAdapter is used to generate an HTML form, the HiddenFields object will define a hidden field for the mode. When the HTML form is submitted, the HTTP request will contain this hidden field value. When executing an action, the mode value is extracted from the HTTP request. If the mode is "Insert", a new row is inserted into the dataset. If the mode is "Edit", a dataset row is updated.

HiddenRecordFields: AdapterHiddenFields

See also HiddenFields, AdapterHiddenFieldsType, Example 10, Example 22

HiddenRecordFields defines the hidden input fields that pass state information needed by each row or record in the HTML form. For example, when TDataSetAdapter is used to generate an HTML form, the HiddenRecordFields object will define a hidden field that identifies a key value for each row in an HTML table. When the HTML form is submitted, the HTTP request will contain these hidden field values. When executing an action that updates multiple rows in a dataset, TDataSetAdapter uses these key values to locate the rows to update.

Mode: text, read/write

See also Example 10

Sets or gets the adapter's mode.

Some adapters support a mode. For example, the TDataSetAdapter supports "Edit", "Insert", "Browse", and "Query" modes. The mode affects the behavior of the adapter. When the TDataSetAdapter is in "Edit" mode, a submitted form updates a row in a table. When the TDataSetAdapter is in "Insert" mode, a submitted form inserts a row in a table.

Name_: text, read

Identifies the variable name of the adapter.

Records: Enumerator, read

See also Example 9

Enumerates the records of the adapter. Use the Records property to loop through the adapter records to generate an HTML table.

AdapterActionType

See also AdapterType, AdapterFieldType

AdapterAction defines the properties and methods of an adapter action.

Properties

Array: Enumerator

See also Example 11

Enumerates the commands of an adapter action. Use the Array property to loop through the commands. Array will be Null if the action does not support multiple commands.

TAdapterGotoPageAction is an example of an action that has multiple commands. This action has a command for each page defined by the parent adapter. The Array property is used to generate a series of hyperlinks so that the end-user can user clicks on a hyperlink to jump to a page.

AsFieldValue: text, read

See also AsHREF, Example 10, Example 21

Provides a text value that can be submitted in a hidden field.

AsFieldValue identifies the name of the action and the action's parameters. Put this value in a hidden field called "__act". When the HTML form is submitted, the adapter dispatcher extracts the value from the HTTP request and uses the value to locate and call the adapter action.

AsHREF: text, read

See also AsFieldValue, Example 11

Provides a text value that can be used as the href attribute value in an <a> tag.

AsHREF identifies the name of the action and the action's parameters. Put this value in an anchor tag to submit a request to execute this action. Note that an anchor tag on an HTML form will not submit the form. If the action makes use of submitted form values then use a hidden form field and AsFieldValue to identify the action.

CanExecute: Boolean, read

Indicates whether the end-user has rights to execute this action.

DisplayLabel: text, read

See also Example 21

Suggests a DisplayLabel for this adapter action.

DisplayStyle: string, read

See also Example 21

Suggests an HTML display style for this action.

Server-side script may use the DisplayStyle to determine how to generate HTML. The built in adapters may return one of the following display styles:

Value	Meaning
''	Undefined display style
'Button'	Display as <input type="submit">
'Anchor'	Use <a>

Enabled: Boolean, read

See also Example 21

Indicates whether this action should be enabled on the HTML page.

Name: string, read

Provides the variable name of this adapter action

Visible: Boolean, read

Indicates whether this adapter field should be visible on the HTML page.

Methods

LinkToPage(PageSuccess, PageFail): AdapterAction, read

See also Example 10, Example 11, Example 21, Page, AdapterActionType

Use LinkToPage to specify pages to display after the action executes. The first parameter is the name of the page to display if the action executes successfully. The second parameter is the name of the page to display if errors occur during execution.

AdapterErrorsType

See also AdapterType.Errors

AdapterErrors defines the properties of an adapter's errors property.

Properties

Field: AdapterField, read

See also AdapterFieldType

Identifies the adapter field that caused an error.

This property will be Null if the error is not associated with a particular adapter field.

ID: integer, read

Provides the numeric identifier for an error.

This property will be zero if an ID is not defined.

Message: text, read

See also Example 7

Provides a text description of the error.

AdapterFieldType

See also AdapterType, AdapterActionType

AdapterField defines the properties and methods of an adapter field.

Properties

CanModify: Boolean, read

See also CanView, AdapterType.CanView

Indicates whether the end-user has rights to modify this field's value.

CanView: Boolean, read

See also CanModify, AdapterType.CanModify

Indicates whether the end-user has rights to view this field's value.

DisplayLabel: text, read

Suggests a DisplayLabel for this adapter field.

DisplayStyle: text, read

See also InputStyle, ViewMode, Example 17

DisplayStyle suggests how to display a read-only representation of a field's value.

Server-side script may use the DisplayStyle to determine how to generate HTML. An adapter field may return one of the following display styles:

Value	Meaning
‘’	Undefined display style
‘Text’	Use <p>
‘Image’	Use <img>. The Image property of the field defines the src property.
‘List’	Use <UL>. Enumerate the Values property to to generate each <LI> item.

The ViewMode property indicates whether to use the InputStyle or DisplayStyle to generate HTML.

DisplayText: text, read

See also EditText, Example 9

Provides text to use when displaying the adapter field's value for reading only. The value of DisplayText may include numeric formatting.

DisplayWidth: integer, read

See also MaxLength

Suggests a display width, in characters, for an adapter field's value.

-1 is returned if the display width is undefined.

EditText: text, read

See also DisplayText, Example 10

Provides text to use when defining an HTML input for this adapter field. The value of EditText is typically unformatted.

Image: AdapterImage, read

See also AdapterImageType, Example 12

Provides an object that defines an image for this adapter field.

Null is returned If the adapter field does not provide an image.

InputStyle: text, read

See also DisplayStyle, ViewMode, Example 17

Suggests an HTML input style for this field.

Server-side script may use the InputStyle to determine how to generate an HTML element. An adapter field may return one of the following input styles:

Value	Meaning
ʻʼ	Undefined input style
'TextInput'	Use <input type=”text”>
'PasswordInput'	Use <input type=”password”>
'Select'	Use <select>. Enumerate the ValuesList property to generate each <option> element.
'SelectMultiple'	Use <select multiple>. Enumerate the ValuesList property to generate each <option> element.
'Radio'	Enumerate the ValuesList property to generate one or more <input type=”radio”>.
'CheckBox'	Enumerate the ValuesList property to generate one or more <input type=”checkbox”>.
'TextArea'	Use <textarea>.
'File'	Use <input type=”file”>

The ViewMode property indicates whether to use the InputStyle or DisplayStyle to generate HTML.

InputName: text, read

See also Example 10

Provides a name for an HTML input element to edit this adapter field.

Use InputName when generating an HTML <input>, <select>, or <textarea> element so that the adapter component will be able to associate between the name/value pairs in the HTTP request with adapter fields.

MaxLength: integer, read

See also DisplayWidth

Indicates the maximum length in characters that can be entered into this field.

-1 is returned if the maximum length is not defined.

Name: text, read

Returns the variable name of the adapter field.

Required: Boolean, read

Indicates whether a value for this adapter field is required when submitting a form.

Value: variant, read

See also Values, DisplayText, EditText

Returns a value that can be used in calculations. For example, use Value when adding two adapter field values together.

Values: AdapterFieldValues, read

See also ValuesList, AdapterFieldValuesType, Value, Example 13

Returns a list of the field's values. The Values property is Null unless this adapter field supports multiple values. A multiple value field would be used, for example, to allow the end-user to select multiple values in a select list.

ValuesList: AdapterFieldValuesList, read

See also Values, AdapterFieldValuesListType, Example 13

Provides a list of choices for this adapter field. Use ValuesList when generating an HTML select list, check box group, or radio button group. Each item in ValuesList has a value and may have a name.

Visible: Boolean, read

Indicates whether this adapter field should be visible on the HTML page.

ViewMode: text, read

See also DisplayStyle, InputStyle, Example 17

Suggests how to display this adapter field value on an HTML page.

An adapter field may return one of the following view modes:

Value	Meaning
‘’	Undefined view mode
'Input'	Generate editable HTML form elements using <input>, <textarea>, or <select>
'Display'	Generate read-only HTML using <p>, , or <img>

The ViewMode property indicates whether to use the InputStyle or DisplayStyle to generate HTML.

Methods

IsEqual(Value): Boolean

See also Example 16

Call this function to compare a variable with an adapter field's value.

AdapterFieldValuesType

See also AdapterFieldType.Values

Provides a list of the field’s values. Multiple value adapter fields support this property. A multiple value field would be used, for example, to allow the end-user to select multiple values in a select list.

Properties

Value: variant, read

See also ValueField

Returns the Value of the current enumeration item.

Records: Enumerator, read

See also Example 15

Enumerates the records in the list of Values.

ValueField: AdapterField, read

See also AdapterFieldType, Example 15

Returns an adapter field for the current enumeration item. Use ValueField, for example, to get the DisplayText for the current enumeration item.

Methods

HasValue(Value): Boolean

See also Example 14

Indicates whether a given value is in the list of field values. This method is to determine whether to select an item in an HTML select list or check an item in a group of check boxes.

AdapterFieldValuesListType

See also AdapterType

Provides a list of possible values for this adapter field.

Use ValuesList when generating an HTML select list, check box group, or radio button group. Each item in ValuesList contains a value and may contain a name.

Properties

Image: AdapterImage, read

Returns the image of the current enumeration item.

Null is returned if the item doesn't have an image.

Records: Enumerator, read

Enumerates the records in the list of Values.

Value: variant, read

Returns the Value of the current enumeration item.

ValueField: AdapterField, read

See also AdapterFieldType, Example 15

Returns an adapter field for the current enumeration item. Use ValueField, for example, to get the DisplayText for the current enumeration item.

ValueName: text, read

Returns the text name of the current item.

Blank is returned if the value does not have a name.

Methods

ImageOfValue(Value): AdapterImage

Looks up the image associated with this value.

Null is returned if there is no image.

NameOfValue(Value): text

Looks up the name associated with this value.

Blank is returned if the value is not found of if the value does not have a name.

AdapterHiddenFieldsType

See also AdapterType.HiddenFields, AdapterType.HiddenRecordFields

Provides access to the hidden field names and values that an adapter requires on HTML forms used to submit changes.

Properties

Name: text, read

Returns the name of the hidden field being enumerated.

Value: text, read

Returns the string value of the hidden field being enumerated.

Methods

WriteFields(Response)

See also Example 10, Example 22

Writes hidden field names and values using <input type="hidden">.

Call this method to write all of the HTML hidden fields to an HTML form.

AdapterImageType

See also AdapterFieldType, AdapterActionType

Represents an image that is associated with an action or a field.

Properties

AsHREF: text, read

See also Example 11, Example 12

Provides a URL that can be used to define an HTML <img> element.

ModuleType

See also Modules

Adapter components can be referenced by name as properties of a module. Also use a module to enumerate the scriptable objects (usually adapters) of a module.

Properties

Name_: text, read

See also Example 20

Identifies the variable name of the module. This is the name used to access the module as a property of the Modules variable.

ClassName_: text, read

See also Example 20

Identifies the VCL class name of the module.

Objects: Enumerator

See also Example 20

Use Objects to enumerate the scriptable objects (typically adapters) within a module.

PageType

See also Page, Example 20

Defines properties and methods of pages.

Properties

CanView: Boolean, read

Indicates whether the end-user has rights to view this page.

A page registers access rights. CanView compares the rights registered by the page with the rights granted to the end-user.

DefaultAction: AdapterAction, read

See also Example 6

Identifies the default adapter action associated with this page.

A default action is typically used when parameters must be passed to a page. DefaultAction may be Null.

HREF: text, read

See also Example 5

Provides a URL that can be used to generate a hyperlink to this page using the <a> tag.

LoginRequired: Boolean, read

Indicates whether the end-user must login before accessing this page.

A page registers a LoginRequired flag. If True then the end-user will not be permitted to access this page unless logged in.

Name: text, read

See also Example 5

Provides the name of the registered page.

If the page is published, the PageDispatcher will generate the page when the page's name is a suffix of the HTTP request's path info.

Published: Boolean, read

See also Example 5

Indicates whether the end-user can access this page by specifying the page name as a suffix to the URL.

A page registers a published flag. The PageDispatcher will automatically dispatch published page. Typically the published flag is used while generating a menu with hyperlinks to pages. Pages that have the published flag set to False are not listed in the menu.

Title: text, read

See also Example 5, Example 18

Provides the title of this page.

The title is typically displayed to the user.

Examples

The following JScript examples demonstrate how many of the server-side scripting properties and methods are used.

Table 29.3 Examples of server-side scripting properties and methods

Example	Description
Example 1	Uses the Application.QualifyFilename method to generate a relative path reference to an image.
Example 2	Declares a variable that references a module.
Example 3	Enumerates the modules in the web application and displays their names in an HTML table.
Example 4	Declares a variable that references a registered page.
Example 5	Enumerates registered pages to generate a menu of hyperlinks to published pages.
Example 6	Enumerates registered pages to generate a menu of hyperlinks to the published pages' default actions.
Example 7	Writes a list of errors detected by an adapter.
Example 8	Enumerates all of the action objects of an adapter to display action object property values in an HTML table.
Example 9	Enumerates the records of an adapter to display adapter field values in an HTML table.
Example 10	Generates an HTML form to edit adapter fields and submit adapter actions.
Example 11	The GotoPage action has an array of commands. The commands are enumerated to generate a hyperlink to jump to each page.
Example 12	Displays an adapter field's image using the <img> tag
Example 13	Displays an adapter field using the <select> and <option> tags.
Example 14	Displays an adapter field as a group of check boxes.
Example 15	Displays an adapter field's values using and tags.
Example 16	Displays an adapter field as a group of radio buttons.
Example 17	Uses the adapter field's DisplayStyle, InputStyle andViewMode properties to generate HTML.

Table 29.3 Examples of server-side scripting properties and methods

Example	Description
Example 18	Uses properties of the Application object and the Page object to generate a page heading.
Example 19	Uses properties of the EndUser object to display the end-user's name, the login command, and the logout command.
Example 20	Lists the scriptable objects in a module.
Example 21	Uses the adapter actions's DisplayStyle property to generate HTML.
Example 22	Generate an HTML table to update multiple detail records.

Example 1

See also Application.Designing , Application.QualifyFileName , Request.PathInfo

This example generates a relative path reference to an image. If the script is in design mode then it references an actual directory; otherwise it references a virtual directory.

```
<%
     function PathInfoToRelativePath(S)
     {
       var R = '';
       var L = S.length
       I = 0
       while (I < L)
       {
         if (S.charAt(I) == '/')
           R = R + '../'
         I++
       }
       return R
     }

     function QualifyImage(S)
     {
       if (Application.Designing)
         return Application.QualifyFileName("..\\images\\" + S);   // relative directory
       else
         return PathInfoToRelativePath(Request.PathInfo) + '../images/' + S;  // virtual
directory
     }
  %>
```

Example 2

See also Modules

This example declares a variable that references WebModule1:

```
<% var M  = Modules.WebModule1 %>
```

Example 3

See also Modules

Example 3 enumerates the instantiated module and displays its variable name and VCL class name in a table:

```
<table border=1>
<tr><th>Name</th><th>ClassName</th></tr>
<%
   var e = new Enumerator(Modules)
   for (; !e.atEnd(); e.moveNext())
   {
%>
  <tr><td><%=e.item().Name_%></td><td><%=e.item()ClassName._%></td></tr>
<%
   }
%>
</table>
```

Example 4

See also Pages, Page.Title

Example 4 declares a variable that references the page named Home. It also displays Home's title.

```
<% var P = Pages.Home %>
<p><%= P.Title %></p>
```

Example 5

See also Pages, Page.Published, Page.HREF, Response.Write

Example 5 enumerates the registered pages and creates a menu displaying hyperlinks to all published pages.

```
<table>
<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>
```

Example 6

See also PageType.DefaultAction

Example 6 enumerates the registered pages and creates a menu displaying hyperlinks to DefaultActions.

```
<table>
<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)
           if (e.item().DefaultAction != null)
             s += '<a href="'+e.item().DefaultAction.AsHREF+'"> +e.item().Title+'</a>'
           else
             s += '<a href="' + e.item().HREF + '">' + e.item().Title + '</a>'
         else
           s += e.item().Title
         c++
       }
     }
     if (c>1) Response.Write(s)
%>
</td>
</table>
```

Example 7

See also AdapterType.Errors, AdapterErrorsType, Modules, Response.Write

Example 7 writes a list of errors detected by an adapter.

```
<% {
   var e = new Enumerator(Modules.CountryTable.Adapter.Errors)
   for (; !e.atEnd(); e.moveNext())
   {
     Response.Write("<li>" + e.item().Message)
   }
   e.moveFirst()
   } %>
```

Example 8

See also AdapterType.Actions, AdapterActionType

This example enumerates all of the actions of an adapter and display action property values in a table.

```
<% // Display some properties of an action in a table
   function DumpAction(A)
   {
%>
      <table border="1">
         <tr><th COLSPAN=2><%=A.Name%></th>
         <tr><th>AsFieldValue:</th><td><%= A.AsFieldValue %></td>
         <tr><th>AsHREF:</th><td><%= A.AsHREF %></span>
         <tr><th>DisplayLabel:</th><td><%= A.DisplayLabel %></td>
         <tr><th>Enabled:</th><td><%= A.Enabled %></td>
         <tr><th>CanExecute:</th><td><span class="value"><%= A.CanExecute %></td>
     </table>
<%
  }
%>

<% // Call the DumpAction function for every action in an adapter
   function DumpActions(A)
   {
     var e = new Enumerator(A)
     for (; !e.atEnd(); e.moveNext())
     {
       DumpAction(e.item())
     }
   }
%>


<%
// Display properties of actions in the adapter named Adapter1
DumpActions(Adapter1.Actions) %>
```

Example 9

See also AdapterType.Records, AdapterFieldType.DisplayText

Example 9 generates an HTML table by enumerating the records of an adapter.

```
<%
// Define some variables that reference the adapter and fields that will be used.

vAdapter=Modules.CountryTable.Adapter
vAdapter_Name=vAdapter.Name
vAdapter_Capital=vAdapter.Capital
vAdapter_Continent=vAdapter.Continent
%>
```

```
<%
// Function to write column text so that all cells have borders
function WriteColText(t)
{
   Response.Write((t!="")?t:" ")
}
%>

<table border="1">
  <tr>
  <th>Name</th>
  <th>Capital</th>
  <th>Continent</th>
<%
   // Enumerate all the records in the adapter and write the field values.

   var e = new Enumerator(vAdapter.Records)
   for (; !e.atEnd(); e.moveNext())
   { %>
      <tr>
      <td><div><% WriteColText(vAdapter_Name.DisplayText) %></div></td>
      <td><div><% WriteColText(vAdapter_Capital.DisplayText) %></div></td>
      <td><div><% WriteColText(vAdapter_Continent.DisplayText) %></div></td>
      </tr>
<%
   }
%>
</table>
```

Example 10

See also AdapterActionType.LinkToPage, AdapterActionType.AsFieldValue, AdapterFieldType.InputName, AdapterFieldType.DisplayText, AdapterType.HiddenFields , AdapterType.HiddenRecordFields

Example 10 generates an HTML form to edit adapter fields and submits adapter actions.

```
<%
// Define some variables that reference the adapter, fields, and actions that will be used.

vAdapter=Modules.CountryTable.Adapter
vAdapter_Name=vAdapter.Name
vAdapter_Capital=vAdapter.Capital
vAdapter_Continent=vAdapter.Continent
vAdapter_Apply=vAdapter.Apply
vAdapter_RefreshRow=vAdapter.RefreshRow

// Put the adapter in "Edit" mode unless the mode is already set.  If the mode is already
// set then this is probably because an adapter action set the mode. For example, an insert
// row action would put the adapter in "Insert" mode.
```

```
if (vAdapter.Mode=="")
  vAdapter.Mode="Edit"
%>
<form name="AdapterForm1" method="post">

  <!-- This hidden field is used to define the action that is executed when the form is
submitted -->

  <input type="hidden" name="__act">


<%
   // Write out hidden fields defined by the adapter.

   if (vAdapter.HiddenFields != null)
   {
     vAdapter.HiddenFields.WriteFields(Response)
   } %>
<% if (vAdapter.HiddenRecordFields != null)
   {
     vAdapter.HiddenRecordFields.WriteFields(Response)
   } %>
  <table>
    <tr>
      <td>
        <table>
          <tr>
          <!-- Write input fields to edit the fields of the adapter -->

          <td>Name</td>
          <td ><input type="text" size="24" name="<%=vAdapter_Name.InputName%>" value="
             <%= vAdapter_Name.EditText %>" ></td>
          </tr>
          <tr>
          <td>Capital</td>
          <td ><input type="text" size="24" name="<%=vAdapter_Capital.InputName%>"
             value="<%= vAdapter_Capital.EditText %>" ></td>
          </tr>
          <tr>
          <td>Continent</td>
          <td ><input type="text" size="24" name="<%=vAdapter_Continent.InputName%>"
             value="<%= vAdapter_Continent.EditText %>" ></td>
          </tr>
        </table>
      </td>
    </tr>
    <tr>
      <td>
        <table>
          <!-- Write submit buttons to execute actions.  LinkToPage is used so that this
               page will be regenerated after executing an action. -->

          <tr>
```

```
          <td><input type="submit" value="Apply"
            onclick = "AdapterForm1.__act.value='
              <%=vAdapter_Apply.LinkToPage(Page.Name).AsFieldValue%>'"></td>
          <td><input type="submit" value="Refresh"
            onclick = "AdapterForm1.__act.value='
              <%=vAdapter_RefreshRow.LinkToPage(Page.Name).AsFieldValue%>'"> </td>
          </tr>
        </table>
      </td>
    </tr>
  </table>
</form>
```

Example 11

See also AdapterActionType.Array, AdapterActionType.AsHREF

Display adapter actions to support paging. The PrevPage, GotoPage, and NextPage actions are displayed as hyperlinks. The GotoPage action has an array of commands. The commands are enumerated to generate a hyperlink to jump to each page.

```
<%
   // Define some variables for adapter and actions

   vAdapter = Modules.WebDataModule1.QueryAdapter
   vPrevPage = vAdapter.PrevPage
   vGotoPage = vAdapter.GotoPage
   vNextPage = vAdapter.NextPage
%>

<!-- Generate a table that will display hyperlinks to move between pages of the adapter -->

<table cellpadding="5">
<tr>
<td>
<%
   // Prevpage displays "<<".  Only use an anchor tag if the action is enabled

   if (vPrevPage.Enabled)
   { %>
     <a href="<%=vPrevPage.LinkToPage(Page.Name).AsHREF%>"><<</a>
<%
   }
   else
   {%>
     <a><<</a>
 <%} %>
<%
   // GotoPage has a list of commands.  Loop through the list.  Only use an anchor tag if
the command
   //  is enabled

   if (vGotoPage.Array != null)
```

```
    {
      var e = new Enumerator(vGotoPage.Array)
      for (; !e.atEnd(); e.moveNext())
      {
%>
        <td>
<%      if (vGotoPage.Enabled)
        { %>
           <a href="<%=vGotoPage.LinkToPage(Page.Name).AsHREF%>">
             <%=vGotoPage.DisplayLabel%></a>
<%      }
        else
        { %>
           <a><%=vGotoPage.DisplayLabel%></a>
<%      }
%>
        </td>
<%
      }
    }
%>
<td>
<%
   // NextPage displays ">>".  Only use an anchor tag if the action is enabled

   if (vNextPage.Enabled)
   { %>
     <a href="<%=vNextPage.LinkToPage(Page.Name).AsHREF%>">>></a>
<%
   }
   else
   {%>
     <a>>></a>
 <%} %>
</td>
</table>
```

Example 12

See also AdapterFieldType.Image

Example 12 displays an adapter field's image.

```
<%
// Declare variables for adapter and field

vAdapter=Modules.WebDataModule3.DataSetAdapter1
vGraphic=vAdapter.Graphic
%>

<!-- Display the adapter field as an image. -->
<img src="<%=(vGraphic.Image!=null) ? vGraphic.Image.AsHREF : ''%>"
alt="<%=vGraphic.DisplayText%>">
```

Example 13

See also AdapterFieldType.Values, AdapterFieldType.ValuesList

Example 13 writes an adapter field with HTML <select> and <option> elements.

```
<%
// Return an object that defines HTML select options for an adapter field.
// The returned object has the following elements:
//
// text - string containing the <option> elements.
// count - the number of <option> elements.
// multiple - string containing the either 'multiple' or ''.  Use this value as an attribute
of the
//           <select> element.
//
// Use as follows:
//   obj=SelOptions(f)
//   Response.Write('<select size="' + obj.count + '" name="' + f.InputName + '" ' +
obj.multiple + '>' +
//     obj.text + '</select>')

function SelOptions(f)
{
   var s=''
   var v=''
   var n=''
   var c=0
   if (f.ValuesList != null)
   {
     var e = new Enumerator(f.ValuesList.Records)
     for (; !e.atEnd(); e.moveNext())
     {
       s+= '<option'
       v = f.ValuesList.Value;
       var selected
       if (f.Values == null)
          selected = f.IsEqual(v)
       else
          selected = f.Values.HasValue(v)
       if (selected)
          s += ' selected'
       n = f.ValuesList.ValueName;
       if (n=='')
       {
         n = v
         v = ''
       }
       if (v!='') s += ' value="' + v + '"'
       s += '>' + n + '</option>\r\n'
       c++
     }
     e.moveFirst()
   }
```

```
   r = new Object;
   r.text = s
   r.count = c
   r.multiple = (f.Values == null) ? '' : 'multiple'
   return r;
 }
%>

<%
   // Generate HTML select options for an adapter field
   function WriteSelectOptions(f)
   {
      obj=SelOptions(f)
%>
      <select size="<%=obj.count%>" name="<%=f.InputName%>" <%=obj.multiple%> >
        <%=obj.text%>
      </select>
<%
   }
%>
```

Example 14

See also AdapterFieldType.Values, AdapterFieldType.ValuesList

Example 14 writes an adapter field as a group of <input type="checkbox"> elements.

```
<%
// Return an object that defines HTML checkboxes for an adapter field.
// The returned object has the following elements:
//
// text - string containing the <input type=checkbox> elements.
// count - the number of <option> elements.
//
// Use as follows to define a checkbox group with three columns and no additional
attributes:
//   obj=CheckBoxGroup(f, 3, '')
//   Response.Write(obj.text)
//
function CheckBoxGroup(f,cols,attr)
{
   var s=''
   var v=''
   var n=''
   var c=0;
   var nm=f.InputName
   if (f.ValuesList == null)
   {
     s+= '<input type="checkbox"'
     if (f.IsEqual(true)) s+= ' checked'
     s += ' value="true"' + ' name="' + nm + '"'
     if (attr!='') s+= ' ' + attr
     s += '></input>\r\n'
```

```
      c = 1
    }
    else
    {
      s += '<table><tr>'
      var e = new Enumerator(f.ValuesList.Records)
      for (; !e.atEnd(); e.moveNext())
      {
        if (c % cols == 0 && c != 0) s += '</tr><tr>'
        s+= '<td><input type="checkbox"'
        v = f.ValuesList.Value;
        var checked
        if (f.Values == null)
          checked = (f.IsEqual(v))
        else
          checked = f.Values.HasValue(v)
        if (checked)
          s+= ' checked'
        n = f.ValuesList.ValueName;
        if (n=='')
          n = v
        s += ' value="' + v + '"' + ' name="' + nm + '"'
        if (attr!='') s+= ' ' + attr
        s += '>' + n + '</input></td>\r\n'
        c++
      }
      e.moveFirst()
      s += '</tr></table>'
    }
    r = new Object;
    r.text = s
    r.count = c
    return r;
  }
%>


<%
   // Write an adapter field as a check box group
   function WriteCheckBoxGroup(f, cols, attr)
   {
      obj=CheckBoxGroup(f, cols, attr)
      Response.Write(obj.text);
   }
%>
```

Example 15

See also AdapterFieldType.Values, AdapterFieldValuesType, AdapterFieldType.ValuesList

This example writes an adapter field as a list of read-only values using <ul> and <li> elements.

```
<%
// Return an object that defines HTML list values for an adapter field.
// The returned object has the following elements:
//
// text - string containing the <li> elements.
// count - the number of elements.
//
// text will be blank and count will be zero if the adapter field does not
// support multiple values.
//
// Use as follows to define a displays a read only list of this an adapter
// fields values.
//   obj=ListValues(f)
//   Response.Write('<ul>' + obj.text + '</ul'>')
//
function ListValues(f)
{
   var s=''
   var v=''
   var n=''
   var c=0;
   r = new Object;
   if (f.Values != null)
   {
     var e = new Enumerator(f.Values.Records)
     for (; !e.atEnd(); e.moveNext())
     {
       s+= '<li>'
       s += f.Values.ValueField.DisplayText;
       s += '</li>'
       c++
     }
     e.moveFirst()
   }
   r.text = s
   r.count = c
   return r;
}
%>

<%
// Write an adapter field as a list of read-only values
function WriteListValues(f)
{
   obj=ListValues(f)
%>
   <ul><%=obj.text%></ul>
<%
}
%>
```

Example 16

See also AdapterFieldValuesListType, AdapterFieldType.Values, AdapterFieldType.ValuesList

Example 16 writes an adapter field as a group of <input type="radio"> elements.

```
<%
// Return an object that defines HTML radiobuttons for an adapter field.
// The returned object has the following elements:
//
// text - string containing the <input type=radio> elements.
// count - the number of elements.
//
// Use as follows to define a radiobutton group with three columns and no additional
attributes:
//   obj=RadioGroup(f, 3, '')
//   Response.Write(obj.text)
//

function RadioGroup(f,cols,attr)
{
   var s=''
   var v=''
   var n=''
   var c=0;
   var nm=f.InputName
   if (f.ValuesList == null)
   {
     s+= '<input type="radio"'
     if (f.IsEqual(true)) s+= ' checked'
     s += ' value="true"' + ' name="' + nm + '"'
     if (attr!='') s+= ' ' + attr
     s += '></input>\r\n'
     c = 1
   }
   else
   {
     s += '<table><tr>'
     var e = new Enumerator(f.ValuesList.Records)
     for (; !e.atEnd(); e.moveNext())
     {
       if (c % cols == 0 && c != 0) s += '</tr><tr>'
       s+= '<td><input type="radio"'
       v = f.ValuesList.Value;
       var checked
       if (f.Values == null)
         checked = (f.IsEqual(v))
       else
         checked = f.Values.HasValue(v)
       if (checked)
         s+= ' checked'
       n = f.ValuesList.ValueName;
       if (n=='')
```

```
        {
        n = v
        }
        s += ' value="' + v + '"' + ' name="' + nm + '"'
        if (attr!='') s+= ' ' + attr
        s += '>' + n + '</input></td>\r\n'
        c++
      }
      e.moveFirst()
      s += '</tr></table>'
    }
    r = new Object;
    r.text = s
    r.count = c
    return r;
  }
%>

<%
   // Write an adapter field as a radiobutton group
   function WriteRadioGroup(f, cols, attr)
   {
      obj=RadioGroup(f, cols, attr)
      Response.Write(obj.text);
   }
%>
```

Example 17

See also AdapterFieldType.DisplayStyle, AdapterFieldType.ViewMode, AdapterFieldType.InputStyle

Example 17 generates HTML for an adapter field based on the field's InputStyle, DisplayStyle, and ViewMode property values.

```
<%
   // Write HTML for an adapter field using the InputStyle, DisplayStyle, and
   // ViewMode properties.
   function WriteField(f)
   {
     Mode = f.ViewMode
     if (Mode == 'Input')
     {
       Style = f.InputStyle
       if (Style == 'SelectMultiple' || Style == 'Select')
          WriteSelectOptions(f)
       else if (Style == 'CheckBox')
         WriteCheckBoxGroup(f, 2, '')
       else if (Style == 'Radio')
         WriteRadioGroup(f, 2, '')
       else if (Style == 'TextArea')
       {
%>
```

```
          <textarea wrap=OFF name="<%=f.InputName%>"><%= f.EditText %></textarea>
<%
        }
        else if (Style == 'PasswordInput')
        {
%>
          <input type="password" name="<%=f.InputName%>"/>
<%
        }
        else if (Style == 'File')
        {
%>
           <input type="file" name="<%=f.InputName%>"/>
<%
        }
        else
        {
%>
          <input type="input" name="<%=f.InputName%>" value="<%= f.EditText %>"/>
<%
        }
      }
      else
      {
        Style = f.DisplayStyle
        if (Style == 'List')
          WriteListValues(f)
        else if (Style == 'Image')
        {
%>
          <img src="<%=(f.Image!=null) ? f.Image.AsHREF : ''%>" alt="<%=f.DisplayText%>">
<%      }
        else
          Response.Write('<p>' + f.DisplayText + '</p>')
      }
    }
%>
```

Example 18

See also Page, Application.Title

This example uses properties of the Application object and Page object to generate a page heading.

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

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

Example 19

See also EndUser

Example 19 uses properties of the EndUser object to display the end-user's name, login command, and logout command.

```
<% 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>
<%   }
   }
%>
```

Example 20

See also ModuleType

This example lists the scriptable objects in a module.

```
<%
   // Write an HTML table list the name and vcl classname of all scriptable objects in a
module
   function ListModuleObjects(m)
   {
%>
     <p></p>
     <table border="1">
     <tr>
     <th colspan="2"><%=m.Name_ + ': ' + m.ClassName_%></th>
     </tr>
<%
     var e = new Enumerator(m.Objects)
     for (; !e.atEnd(); e.moveNext())
     {
%>
     <tr>
     <td>
        <%= e.item().Name_ + ': ' + e.item().ClassName_ %>
```

```
     </td>
     </tr>
<%
     }
%>
     </table>
<%
   }
%>
```

Example 21

See also AdapterActionType.DisplayStyle, AdapterActionType.Enabled,

Example 21 generates HTML for an adapter action based on the actions's DisplayStyle property.

```
<%
   // Write HTML for an adapter action using the DisplayStyle property.
   //
   // a - action
   // cap - caption.  If blank the action's display label is used.
   // fm - name of the HTML form
   // p - page name to goto after action execution.  If blank, the current page is used.
   //
   // Note that this function does not use the action's Array property.   Is is assumed that
   // the action has a single command.
   //
   function WriteAction(a, cap, fm, p)
   {
       if (cap == '')
         cap = a.DisplayLabel
       if (p == '')
         p = Page.Name
       Style = a.DisplayStyle
       if (Style == 'Anchor')
       {

          if (a.Enabled)
          {
            // Do not use the href property.  Instead, submit the form so that  HTML form
            // fields are part of the HTTP request.
%>
            <a href=""onclick="<%=fm%>.__act.value='
               <%=a.LinkToPage(p).AsFieldValue%>';<%=fm%>.submit();return false;">
               <%=cap%></a>
<%
          }
          else
          {
%>
            <a><%=cap%></a>
<%
```

```
          }
       }
       else
       {
%>
          <input type="submit" value="<%= cap%>"
onclick="<%=fm%>.__act.value='<%=a.LinkToPage(p).AsFieldValue%>'">
<%
       }
   }
%>
```

Example 22

See also AdapterType.HiddenFields, AdapterType.HiddenRecordFields, AdapterType.Mode

This example generates an HTML table to update multiple detail records.

```
<%
vItemsAdapter=Modules.DM.ItemsAdapter
vOrdersAdapter=Modules.DM.OrdersAdapter
vOrderNo=vOrdersAdapter.OrderNo
vCustNo=vOrdersAdapter.CustNo
vPrevRow=vOrdersAdapter.PrevRow
vNextRow=vOrdersAdapter.NextRow
vRefreshRow=vOrdersAdapter.RefreshRow
vApply=vOrdersAdapter.Apply
vItemNo=vItemsAdapter.ItemNo
vPartNo=vItemsAdapter.PartNo
vDiscount=vItemsAdapter.Discount
vQty=vItemsAdapter.Qty
%>

<!-- Use two adapters to update multiple detail records.
     The orders adapter is associated with the master dataset.
     The items adapter is associated with the detail dataset.
     Each row in a grid displays values from the items adapter.  One
     column display an <input> element for editing Qty.  The apply button
     updates the Qty value in each detail record.

<!-- Display the order number and customer number values -->
<h2>OrderNo: <%= vOrderNo.DisplayText %></h2>
<h2>CustNo: <% vCustNo.DisplayText %></h2>

<%
   // Put the items adapter in edit mode because this form updates
   // the Qty field.
   vItemsAdapter.Mode = 'Edit'
%>

<form name="AdapterForm1" method="post">
```

```
  <!-- Define a hidden field for submitted the action name and parameters -->
  <input type="hidden" name="__act">

<%
   // Write hidden fields containing state information about the
   // orders adapter and items adapter.
   if (vOrdersAdapter.HiddenFields != null)
     vOrdersAdapter.HiddenFields.WriteFields(Response)
   if (vItemsAdapter.HiddenFields != null)
     vItemsAdapter.HiddenFields.WriteFields(Response)



   // Write hidden fields containing state information about the current
   // record of the orders adapter.
   if (vOrdersAdapter.HiddenRecordFields != null)
     vOrdersAdapter.HiddenRecordFields.WriteFields(Response)%>

  <table border="1">
    <tr>
    <th>ItemNo</th>
    <th>PartNo</th>
    <th>Discount</th>
    <th>Qty</th>
  </tr>
<%
   var e = new Enumerator(vItemsAdapter.Records)
   for (; !e.atEnd(); e.moveNext())
   { %>
     <tr>
     <td><%=vItemNo.DisplayText%></td>
     <td><%=vPartNo.DisplayText%></td>
     <td><%=vDiscount.DisplayText%></td>
     <td><input type="text" name="<%=vQty.InputName%>" value="<%= vQty.EditText %>" ></td>
     </tr>
<%
     // Write hidden fields containing state information about each record of the
     // items adapter.   This is needed by the items adapter when updating the Qty field.

     if (vItemsAdapter.HiddenRecordFields != null)
       vItemsAdapter.HiddenRecordFields.WriteFields(Response)
   }
%>
  </table>
  <p></p>
  <table>
    <td><input type="submit" value="Prev Order"

onclick="AdapterForm1.__act.value='<%=vPrevRow.LinkToPage(Page.Name).AsFieldValue%>'"></td>
    <td><input type="submit" value="Next Order"

onclick="AdapterForm1.__act.value='<%=vNextRow.LinkToPage(Page.Name).AsFieldValue%>'"></td>
    <td><input type="submit" value="Refresh"
```

```
onclick="AdapterForm1.__act.value='<%=vRefreshRow.LinkToPage(Page.Name).AsFieldValue%>'"></
td>
    <td><input type="submit" value="Apply"
      onclick="AdapterForm1.__act.value='<%=vApply.LinkToPage(Page.Name).AsFieldValue%>'"></
td>
  </table>
</form>
```

Dispatching requests

When the WebSnap application receives an HTTP request message, it creates a *TWebRequest* object to represent the HTTP request message, and a *TWebResponse* object to represent the response that should be returned.

Web context

Before handling the request, the Web application module initializes the Web context object (of type *TWebContext*). The Web context object, which is accessed through the global WebContext function, is a thread variable that provides global access to variables used by components servicing the request. For example, the Web context contains the *TWebResponse* and *TWebRequest* objects, as well as the adapter request and adapter response objects discussed later in this section.

Dispatcher components

The dispatcher components within the Web Application module control the flow of the application. The dispatchers determine how to handle certain types of HTTP request messages by examining the HTTP request.

The adapter dispatcher component (*TAdapterDispatcher)* looks for a content field, or a query field, that identifies an adapter action component or an adapter image field component. If the adapter dispatcher finds a component, it will pass control to that component.

The Web dispatcher component (*TWebDispatcher*) maintains a collection of action items (*TWebActionItem*) that know how to handle certain types of HTTP request messages. The Web dispatcher looks for an action item that matches the request. If it finds one, it passes control to that action item. The Web dispatcher also looks for auto-dispatching components that can handle the request.

The page dispatcher component (*TPageDispatcher*) examines the pathInfo property of the *TWebRequest* object, looking for the name of a registered Web page module. If the dispatcher finds a Web page module name, it passes control to that Web page module.

Adapter dispatcher operation

The adapter dispatcher component automatically handles HTML form submissions, and requests for dynamic images, by calling adapter action and field components.

Using adapter components to generate content

In order for WebSnap applications to automatically execute adapter actions and retrieve dynamic images from adapter fields, the HTML content must be properly constructed. If the HTML content is not properly constructed, then the resulting HTTP request will not contain the information that the adapter dispatcher needs to call adapter action and field components.

To reduce errors in constructing the HTML page, adapter components indicate what the names and values of HTML elements must be. Adapter components have methods that retrieve the names and values of hidden fields that must appear on an HTML form used to update adapter fields. Typically, page producers use server-side scripts to retrieve names and values from adapter components and generates HTML using these names and values. For example, the following script constructs an <IMG> element that references the field called Graphic from Adapter1:

```
<img src="<%=Adapter1.Graphic.Image.AsHREF%>" alt="<%=Adapter1.Graphic.DisplayText%>">
```

When the Web application evaluates the script, the HTML src attribute will contain the information necessary to identify the field and any parameters that the field component needs to retrieve the image. The resulting HTML might look like this:

```
<img src="?_lSpecies No=90090&__id=DM.Adapter1.Graphic" alt="(GRAPHIC)">
```

When the browser sends an HTTP request to retrieve this image to the Web application, the adapter dispatcher will be able to determine that the Graphic field of Adapter1, in the module DM, should be called with the parameter "Species No=90090". The adapter dispatcher will call the Graphic field to write an appropriate HTTP response.

The following script constructs an <A> element referencing the EditRow action of Adapter1 and the page called "Details":

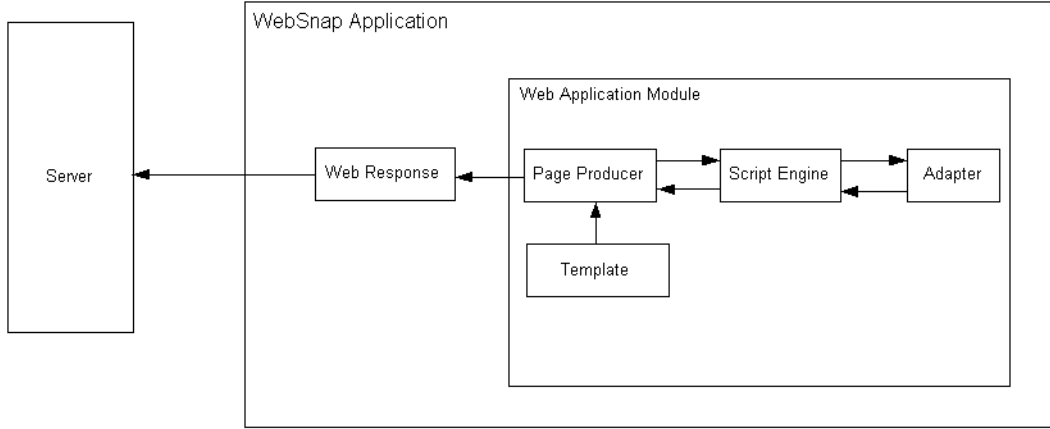
```
<a href="<%=Adapter1.EditRow.LinkToPage("Details", Page.Name).AsHREF%>">Edit...</a>
```

The resulting HTML might look like this:

```
<a href="?&_lSpecies No=90310&__sp=Edit&__fp=Grid&__id=DM.Adapter1.EditRow">Edit...</a>
```

When the end-user clicks this hyperlink and the browser sends an HTTP request, the adapter dispatcher will be able to determine that the EditRow action of Adapter1, in the module DM, should be called with the parameter "Species No=903010". The adapter dispatcher will also indicate that the Edit page is to be displayed if the action executes successfully, and that the Grid page is to be displayed if action execution fails. It will then call the EditRow action to locate the row to be edited, and the page named Edit will be called to generate an HTTP response. Figure 29.10 shows how adapter components are used to generate content.

Figure 29.10 Generating content flow



Adapter requests and responses

When the adapter dispatcher receives the client request, the adapter dispatcher creates adapter request and adapter response objects to hold information about the HTTP request. The adapter request and adapter response objects are stored in the Web context to allow access during the processing of the request.

The adapter dispatcher creates two types of adapter request objects: action and image. It creates the action request object when executing an adapter action. It creates the image request object when retrieving an image from an adapter field.

The adapter response object is used by the adapter component to indicate the response to an adapter action or adapter image request. There are two types of adapter response objects, action and image.

Action request

The action request object is responsible for breaking the HTTP request down into information needed to execute an adapter action. The types of information needed for executing an adapter action may include:

- **Component Name**—Identifies the adapter action component.
- **Adapter Mode**—Adapters can define a mode. For example, *TDataSetAdapter* supports Edit, Insert, and Browse modes. An adapter action may execute differently depending on the mode. For example, the *TDataSetAdapter* Apply action adds a new record when in Insert mode, and updates a record when in Edit mode.
- **Success Page**—The success page identifies the page displayed after successful execution of the action.
- **Failure Page**—The failure page identifies the page displayed if an error occurs during execution of the action.

- **Action Request Parameters**—This identifies the parameters need by the adapter action. For example, the *TDataSetAdapter* Apply action will include the key values identifying the record to be updated.
- **Adapter Field Values**—These are the values for the adapter fields passed in the HTTP request when an HTML form is submitted. A field value can include new values entered by the end-user, the original values of the adapter field, and uploaded files.
- **Record Keys**—If an HTML form submits changes to multiple records, keys used by the adapter action component are required to uniquely identify each record so that the adapter action can be performed on each record. For example, when the *TDataSetAdapter* Apply action is performed on multiple records, the record keys are used to locate each record in the dataset before updating the dataset fields.

Action response

The Action Response object generates an HTTP response on behalf of an adapter action component. The adapter action indicates the type of response by setting properties within the object, or by calling methods in the Action Response object. The properties include:

- *Redirect Options*—The redirect options indicate whether to perform an HTTP redirect instead of returning HTML content.
- *Execution Status*—Setting the status to success causes the default action response to be the content of the success page identified in the Action Request.

The Action response methods include:

- *RespondWithPag* —The adapter action calls this method when a particular Web page module should generate the response.
- *RespondWithComponent*—The adapter action calls this method when the response should come from the Web page module containing this component.
- *RespondWithURL*—The adapter action calls this method when the response is a redirect to a specified URL.

When responding with a page, the Action response object attempts to use the page dispatcher to generate page content. If it does not find the page dispatcher, it calls the Web Page module directly.

Figure 29.13 illustrates how action request and action response objects handle a request.

Figure 29.11 Action request and response

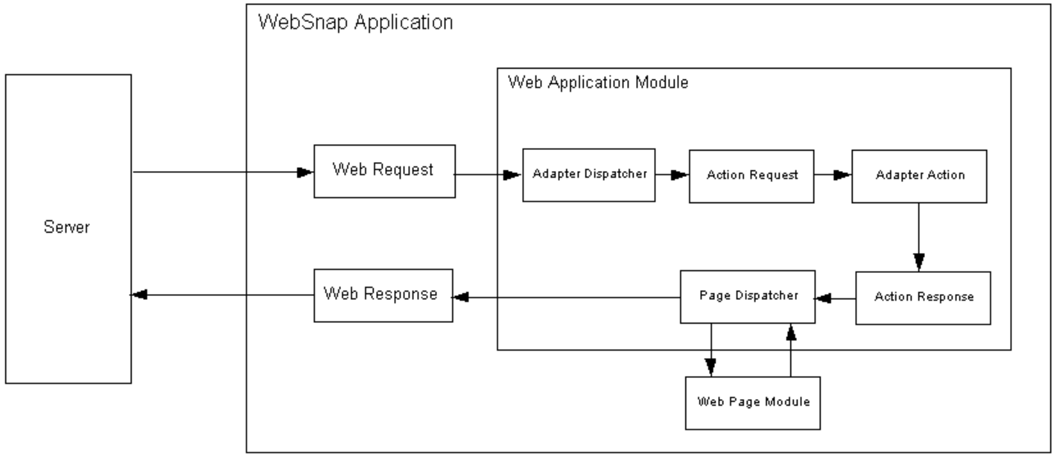


Image request

The Image Request object is responsible for breaking the HTTP request down into the information required by the adapter image field to generate an image. The types of information represented by the Image Request include:

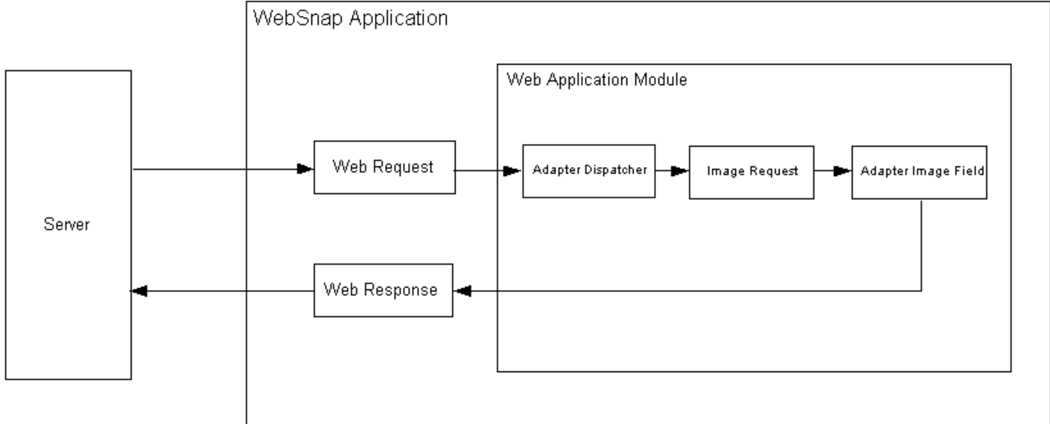
- Component Name - Identifies the adapter field component.
- Image Request Parameters - Identifies the parameters needed by the adapter image. For example, the *TDataSetAdapterImageField* object needs key values to identify the record that contains the image.

Image response

The Image response object contains the *TWebResponse* object. Adapter fields respond to an adapter request by writing an image to the Web response object.

Figure 29.12 illustrates how adapter image fields respond to a request.

Figure 29.12 Image response to a request



Dispatching action items

The Web dispatcher (*TWebDispatcher*) searches through its list of action items for one that:

- matches the Pathinfo portion of the target URL's request message, and
- can provide the service specified as the method of the request message.

It accomplishes this by comparing the *PathInfo* and *MethodType* properties of the *TWebRequest* object with the properties of the same name on the action item.

When the dispatcher finds the appropriate action item, it causes that action item to fire. When the action item fires, it does one of the following:

- Fills in the response content and sends the response, or signals that the request has been completely handled.
- Adds to the response, and then allows other action items to complete the job.
- Defers the request to other action items.

After the dispatcher has checked all of its action items, if the message has not been handled correctly, the dispatcher checks for specially registered auto-dispatching components that do not use action items. These components are specific to multi-tiered database applications. If the request message has still not been fully handled, the dispatcher calls the default action item. The default action item does not need to match either the target URL or the method of the request.

Page dispatcher operation

When the page dispatcher recieves a client request, it determines the page name by checking the Pathinfo portion of the target URL's request message. If the pathinfo

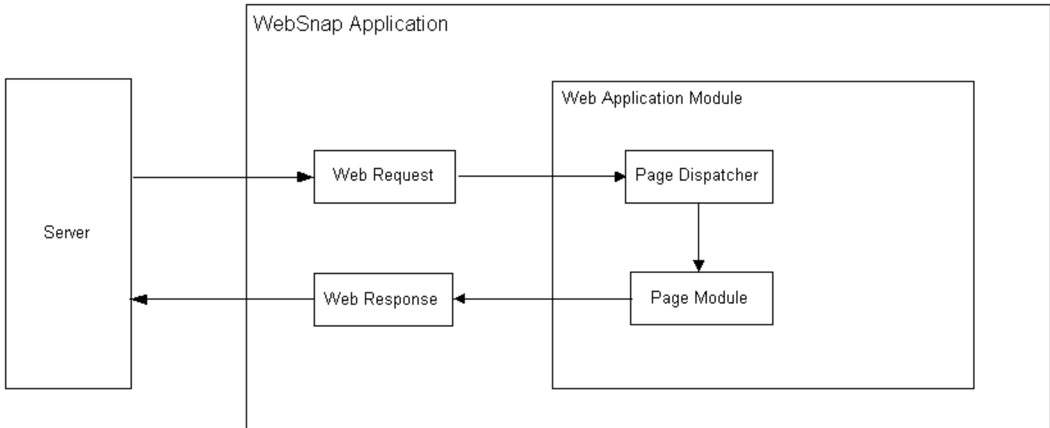
portion is not blank, the page dispatcher uses the ending word of pathinfo as the page name. If the pathinfo portion is blank, the page dispatcher tries to determine a default page name.

If the page dispatcher's *DefaultPage* property contains a page name, then the page dispatcher uses this name as the default page name. If the *DefaultPage* property is blank, and the Web application module is a page module, then the page dispatcher uses the name of the Web application module as the default page name.

If the page name is not blank, the page dispatcher searches for a Web page module with a matching name. If it finds a Web page module, then it calls that module to generated a response. If the page name is blank, or if the page dispatcher does not find a Web page module, the page dispatcher raises an exception.

Figure 29.13 shows how the page dispatcher responds to a request.

Figure 29.13 Dispatching a page



Index

A

action items
    Web dispatchers 72
action requests
    HTML 69
action responses 70
actions
    Web adapters 6
active scripting 27
adapter components
    using 68
adapter dispatcher requests 69
adapter dispatchers 9, 10, 67
adapter page producers 30
AdapterPageProducer 11
adapters 2, 5 to 6
    actions 6
    errors 6
    fields 5
    records 6
Apache applications
    creating 8
Apache server DLLs
    creating 8
application adapters 10
applications
    Apache 8
    CGI stand-alone 8
    ISAPI 8
    NSAPI 8
    Web server 7
    Win-CGI stand-alone 8

C

CGI applications
    creating 8
components
    dispatcher 67
    page producers 4
creating a Web page module 19

D

data modules
    Web 2, 3, 4 to 5
debugging
    Web server applications 9
DefaultPage property 73
dispatch actions 10
dispatcher components 67
    adapters 68
dispatchers
    action items 72
dispatching requests
    WebSnap 67

E

editing script 28
end user adapters 10
errors
    Web adapters 6

F

factory 5
fields
    Web adapters 5

H

HTML Result tab 2
HTML Script tab 2
HTML templates 4
HTTP request messages 67
HTTP requests
    images 71
HTTP responses
    actions 70
    images 71

I

image requests 71
ISAPI applications
    creating 8

M

Microsoft Server DLLs
    creating 8

N

Netscape Server DLLs
    creating 8
NSAPI applications
    creating 8

O

objects
    scripting 29

P

page dispatchers 10, 67
    dispatchers
        page 72
page modules 2, 3, 3 to 4
page producers 2, 4, 6 to 7, 30
    components 4
    templates 4
    types 11

R

records
    Web adapters 6
request
    actions and HTML 69
requests
    adapters 69
    dispatching 67
    images 71
responses
    actions 70
    adapters 69
    images 71
rows
    Web adapters 6

S

script objects 29
scripting 7
    server-side 26 to 67
scripts
    active 27
    editing and viewing 28
    generating in WebSnap 28
server types 8
servers
    Web application debugger 9
server-side 7
server-side scripting 7, 26 to 67
server-side scripting
  reference 30 to 67
sessions services 10

T

TAdapterDispatcher 67
TAdapterPageProducer 28
templates
    page producers 4

test server, Web Application Debugger 9
TPageDispatcher 67
tutorial
  WebSnap 11 to 23
TWebAppDataModule 2
TWebAppPageModule 2
TWebContext 67
TWebDataModule 2
TWebDispatcher 67, 72
TWebPageModule 2
types
  Web modules 2
types of Web servers 8

U

user list services 10

V

viewing scripts 28

W

Web 11
Web adapters
  actions 6
  errors 6
  fields 5
  records 6
Web application debugger 9
Web application modules 2, 3
Web context 67
Web data modules 2, 3, 4 to 5
  structure 5
Web dispatchers 67
  action items 72
Web modules 2, 2 to 5
  types 2
Web page modules 2, 3, 3 to 4
Web scripting 7
Web servers
  types 8
WebSnap
  global script objects 31
  scripting object types 35
  server-side scripting 26 to 67
  server-side scripting examples 47
  server-side scripting reference 30 to 67
WebSnap tutorial 11 to 23
Win-CGI programs
  creating 8

X

XSLPageProducer 4