Internet-Enabling Your Applications with WebBroker

by Nick Hodges

CHAPTER

31

IN THIS CHAPTER

- ISAPI, NSAPI, and CGI Web Server Extensions 1645
- Creating Web Applications with Delphi 1647
- Dynamic HTML Pages with HTML Content Producers 1655
- Maintaining State with Cookies 1665
- Redirecting to a Different Web Site 1670
- Retrieving Information from HTML Forms 1671
- Data Streaming 1673
- Summary 1677

The Internet's popularity has exploded, and its use by computer owners has become almost a given. The technology that makes the Internet work is deceptively simple, and as a result, many business organizations are using the technology to create *intranets*—small Web networks accessible only to those within a given organization. Intranets are proving to be an inexpensive and highly effective way to leverage an organization's information systems. As new technologies arrive, some intranets are even being expanded to *extranets*—networks that allow limited access but are not limited to an organization's boundaries.

All of this, of course, makes programming for the Internet/intranet a very important arrow in a programmer's quiver. As you might expect, Delphi makes programming for the Internet/intranet a very straightforward task. Delphi lets you bring its full power to the Web in the following ways:

- By encapsulating the Hypertext Transfer Protocol (HTTP) in easily accessible objects
- By providing an application framework around the application programming interfaces (APIs) of the most popular and powerful Web servers
- By providing a *Rapid Application Development* (RAD) approach to building Web server extensions

With Delphi and its WebBroker components, you can easily build Web server extensions that provide customized, dynamic Hypertext Markup Language (HTML) pages that include access to data from virtually any source.

TIP

The WebBroker components are provided as a part of Delphi Enterprise. If you are a Delphi Professional user, you can purchase the WebBroker components as a separate add-on. Visit the Borland Web site (`http://www.borland.com`) for more information.

The basic technology that makes the Web possible is quite simple. The two agents in the process—the Web client, or client, and the Web server—must establish a communications link and pass information to and from each other. The client requests information and the server provides it. Of course, the client and the server have to agree on how to communicate and what form the information they share will take. They do this across the Web with nothing more than an ASCII byte stream. The client sends a text request and gets a text answer back. The client knows little about what takes place on the server. This simple process allows for cross-platform communication, normally by means of the TCP/IP protocol.

The standard method of communicating used on the Web is the Hypertext Transfer Protocol (HTTP). A *protocol* is simply an agreement about a way of doing business, and HTTP is a protocol designed to pass information from the client to the server in the form of a request, and

from the server to the client in the form of a response. It does so by formatting information as a byte stream of ASCII characters and sending this information between the two agents. The HTTP protocol itself is both flexible and powerful. When used in concert with Hypertext Markup Language (HTML), it can quickly and easily provide Web pages to a browser.

An HTTP request might look like this:

```
GET /mysite/webapp.dll/dataquery?name=CharlieTuna&company=Borland HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.mysite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

HTTP is *stateless*, which means that the server has no knowledge of the state of the client and that the communication between the server and the client ends when the request has been satisfied. This makes creating database applications using HTTP somewhat problematic because many database applications rely on the client having access to a live dataset. State information can be stored through the use of *cookies*—pieces of information that are stored on the client as a result of the HTTP response. Cookies are discussed later in the chapter.

ISAPI, NSAPI, and CGI Web Server Extensions

Web servers are the engines that make the Web function. They provide all the content to Web browsers, whether that content is HTML pages, Java applets, or ActiveX controls. Web servers are the tools that provide responses to a client's request. Many different Web servers are available for use on any of the different popular platforms.

The Common Gateway Interface

The first Web servers could merely retrieve and return an existing, static HTML page. Web site managers could provide nothing more in a Web site than the pages that were present on the server at the time of the request. Soon, however, it became clear that a higher level of interaction between client and server was required, and the *Common Gateway Interface* (CGI) was developed as a result. CGI allowed the Web server to launch a separate process based on input from the user, work on that information, and return a dynamically created Web page to the client. A CGI program could do any type of data manipulation that the programmer required, and it could return any sort of page that HTML would allow.

Standard CGI applications work by reading from `STDIN`, writing to `STDOUT`, and reading environmental variables. WinCGI works by storing the request parameters in a file, launching the WinCGI application, reading and processing the data in the file, and then writing an HTML file, which is then returned by the Web server. Suddenly, the Web took a large step forward, because servers could now provide tailored, unique responses to users' requests.

However, CGI and WinCGI applications have some drawbacks. Each request must launch its own process on the server, so multiple requests can easily tie up even a moderately busy server. The task of creating a file, launching a separate process, executing the process, and then writing and returning yet another file is relatively slow.

ISAPI and NSAPI

The major Web server vendors, Microsoft and Netscape, saw the weaknesses inherent in CGI programming, but they also saw the advantages of dynamic Web creation. Therefore, instead of using a separate process for each request, each company wrote APIs for its Web servers that allowed Web server extensions to be run as *dynamic link libraries* (DLLs). DLLs can be loaded once and then respond to any number of requests. They run as part of the Web server process, executing their code in the same memory space as the Web server itself. Instead of having to pass information back and forth as files, Web server extensions can simply pass the information back and forth inside the same memory space. This allows for faster, more efficient, and less resource-intensive Web applications.

Microsoft provides the rather simple and straightforward *Internet Server Application Programming Interface* (ISAPI) with its Internet Information Server (IIS), and Netscape provides the more complex *Netscape Application Programming Interface* (NSAPI) with its family of Web servers.

Delphi provides access to both APIs through the `NSAPI.PAS` and `ISAPI.PAS` units. To run the applications in this chapter, you have to be running an IIS server, a Netscape server, or one of a number of shareware or freeware servers that meet the ISAPI specification.

TIP

If you do not currently have a Web server installed, you can download the Microsoft Personal Web Server from Microsoft's Web site (`http://www.microsoft.com`). It is freeware and is ISAPI-compliant. It will run all the examples in this chapter.

Using Web Servers

Whichever Web server you are using, you should bear in mind several things when running Web server applications. First of all, because the extensions are DLLs, they will be loaded into memory and remain in memory while the Web server is running. Therefore, if you are building and testing applications with Delphi, you may have to shut down the Web server to recompile the application because Windows will not allow you to rewrite a file that is being executed. This may vary between Web

servers, but it is true for the Microsoft Personal Web Server. In addition, Web servers generally require that you select a base directory on your system as the root directory for all your HTML files. You can tell Delphi to send your Web applications directly to that directory by entering the full path of the directory into the Project, Options, Directories/Conditionals Output Directory combo box. Finally, you can even debug your Web applications while they are running. Delphi's documentation includes instructions on how to do this. These instructions can be found in the online help under ISAPI, Debugging. The Web server is used as the host application. Each of the major Web servers is configured a bit differently, so check your server's documentation and the Delphi documentation mentioned for further information.

Creating Web Applications with Delphi

Delphi's WebBroker components make developing Internet/intranet applications easy. The following sections discuss these components and how they allow you to focus on the content of your Web servers without having to worry about the details of HTTP communications protocols.

TWebModule and TWebDispatcher

If you select File, New from the Delphi menu, the New Items dialog box appears. Select the Web Server Application icon to open a wizard that will allow you to select the type of Web server extension. The three choices are ISAPI/NSAPI, CGI, and WinCGI applications. This chapter deals with the ISAPI/NSAPI application type. The construction of the CGI server extensions is done in almost the same manner; however, the ISAPI applications are easier to deal with and run.

NOTE

Delphi also includes a project, `ISAPITER.DPR`, that allows you to run ISAPI modules on an NSAPI-based Web server. The online help has information on how to set up a Netscape Web server to run the ISAPI DLLs created in this chapter.

After you select the application type, Delphi creates a project based on a `TWebModule`. The main project itself is a DLL, and the main unit contains the `TWebModule`. `TWebModule` is a descendant of `TDataModule`, and it contains all the logic needed to receive the HTTP request and respond to it. A `TWebModule` can accept only nonvisual controls, just like its ancestor. You can use all the database controls, as well as the controls on the Internet page of the Component

Palette that produce HTML, to produce content in a `TWebModule`. This allows you to add business rules for your Web-based application in the same manner as you can with `TDataModule` in regular applications.

The `TWebModule` has an `Actions` property, which contains a collection of `TWebActionItem` objects. A `TWebActionItem` allows you to execute code based on a given request. Each `TWebActionItem` has its own name; when a client makes a request based on that name, your code is executed and the appropriate response is given.

NOTE

You can create a Web server application with one of your existing data modules. The `TWebModule` has as one of its fields the `TWebDispatcher` class. This class is included on the Component Palette as the `TWebDispatcher` component. If you replace the default `TWebModule` in your Web server application with an existing data module by using the Project Manager, you can drop a `TWebDispatcher` component on it and it will become a Web server application. The `TWebDispatcher` component on the Internet page of the Component Palette adds all the functionality encapsulated in the `TWebModule`. So if you have all your business rules wrapped up in an existing `TDataModule`, making those rules available to your Web applications is as easy as pointing and clicking. A `TDataModule` with a `TWebDispatcher` component is functionally equivalent to a `TWebModule`. The only difference is that you access the HTTP actions through the `TWebDispatcher` component and not the `TDataModule` itself.

Select the `TWebModule` so that its properties are displayed in the Object Inspector. Select the `Actions` property and either double-click it or select the property editor with the small ellipsis (…) button. This will bring up the `WebModule` Actions dialog. Click the New button and select the resulting `WebActionItem` in the property editor that appears. The `Action` item's properties will then be displayed in the Object Inspector. Go to the `PathInfo` property and enter */test*. Then go to the Events page in the Object Inspector and double-click the `OnAction` event to create a new event handler. It will look like this:

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

end;
```

This event handler contains all the information about the request that generated this action and the means to respond to it. The client's request information is contained in the `Request` parameter, which is of type `TWebRequest`. The `Response` parameter is of type `TWebResponse`, and it

is used to send the necessary information back to the client. Within this event handler, you can write any code necessary to respond to the request, including file manipulation, database actions, and anything else needed to send an HTML page back to the client.

Before we get into the depths of the `TWebModule`, a simple example will help demonstrate the basics of how a Web server application works. The simplest way to create an HTML page that responds to the client's request is to build the HTML on the fly. You can do this easily by using a `TStringList`. After the HTML is placed into the `TStringList`, it can easily be assigned to the `Content` property of the `Response` parameter. `Content` is a string, and it is used to hold the HTML to be returned to the client. This is the only property of `Response` that must be filled because it contains the data to be displayed. If it is left blank, the client's browser will report that the requested document is empty. Listing 31.1 shows the code that you must add to the `/test` action item event handler.

LISTING 31.1 The `WebModule1WebActionItem1Action` Event Handler

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
   Page: TStringList;
begin
   Page := TStringList.Create;
   try
     with Page do
     begin
          Add('<HTML>');
          Add('<HEAD>');
      Add('<TITLE>Web Server Application -- Basic Sample</TITLE>');
          Add('</HEAD>');
          Add('<BODY>');
          Add('<B>This page was created on the fly by Delphi</B><P>');
          Add('<HR>');
          Add('See how easy it was to create a page on the fly with Delphi''s
    ➥Web Extensions?');
          Add('</BODY>');
          Add('</HTML>');
     end;
          Response.Content := Page.Text;
   finally
     Page.Free;
   end;
   Handled := True;
end;
```

Save the project as SAMPLE1.DLL, compile it, and place the resulting file in the default directory for your ISAPI- or NSAPI-capable Web server. Then, point your browser to the following location:

```
<web server address>/sample1.dll/test
```

You should see the expected Web page in your browser, as shown in Figure 31.1.

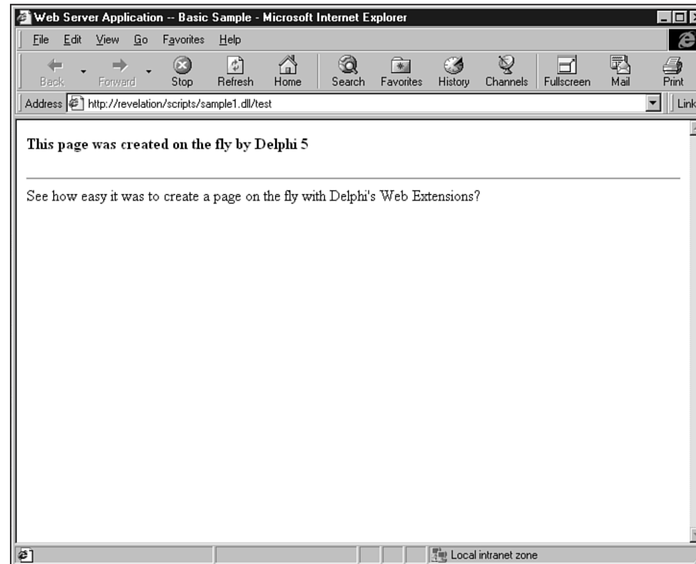

FIGURE 31.1
A sample Web page.

NOTE

If you take Listing 31.1's code from the CD-ROM accompanying this book and place it on your computer, maintaining the same directory structure as on the CD-ROM, you can easily set your Web server up to access the HTML and the DLLs to run all the sample applications from this chapter. Simply create a virtual Web server directory for the root directory and an ISAPI-capable directory that points to the \bin directory. Then, you can open up the INDEX.HTM file in the root directory, giving you access to all the sample code. Note that if you copy the files from the CD-ROM, they will have the read-only flag set. You will have to remove that flag in Explorer if you want to edit the files copied from the CD-ROM.

Note that the result of the project's compilation is a DLL that conforms to the ISAPI specification. The project's source code reveals the following:

```
library Sample1;
uses
  WebBroker,
  ISAPIApp,
  Unit1 in 'Unit1.pas' {WebModule1: TWebModule};
{$R *.RES}
exports
  GetExtensionVersion,
  HttpExtensionProc,
  TerminateExtension;
begin
  Application.Initialize;
  Application.CreateForm(TWebModule1, WebModule1);
  Application.Run;
end.
```

Note the three exported routines. These three—`GetExtensionVersion`, `HttpExtensionProc`, and `TerminateExtension`—are the only three procedures required by the ISAPI specification.

CAUTION

Like a typical application, your ISAPI application uses a global `Application` object. However, unlike a regular application, this project does not use the `Forms` unit. Instead, the `WebBroker` unit contains an `Application` variable declared as type `TWebApplication`. It handles all the special calls needed to be able to hook into an ISAPI- or NSAPI-capable Web server. As a result, you should never try to add the `Forms` unit to an ISAPI-based Web server extension because this may confuse the compiler into using the wrong `Application` variable.

This simple project illustrates how easy it is to build a Web server application and provide a response to a client's request by using Delphi. This was a relatively simple example, creating HTML dynamically in code. However, as you will soon see, Delphi provides the tools to respond in much more complex and interesting ways. Before looking at what Delphi can do in this regard, we will delve a little deeper into the workings of a WebBroker application in the following section.

TWebRequest and TWebResponse

`TWebRequest` and `TWebResponse` are abstract classes that encapsulate the HTTP protocol. `TWebRequest` provides access to all the information passed to the server by the client, and

TWebResponse contains properties and methods that allow you to respond in any of the multiple ways that the HTTP protocol allows. Both of these classes are declared in the HTTPAPP.pas unit, which is used by the WebBroker.pas unit. ISAPI-based Web applications actually use TISAPIResponse and TISAPIRequest, which are descendants of the abstract classes and are declared in ISAPIAPP.PAS. The power of polymorphism allows Delphi to pass the TISAPI*xxx* classes to the TWeb*xxx* parameters of the OnAction event handler in TWebModule.

TISAPIRequest contains all the information passed by a client when making a request for a Web page. You can gather information about the client from the request. Many of the properties may be blank for any given request, because not all fields are completed for every HTTP request. The RemoteHost and RemoteAddr properties contain the IP address of the requesting machine. The UserAgent property contains information about the browser that the client is using. The Accept property includes a listing of the types of graphics that the user's browser can display. The Referer property contains the URL for the page that the user clicked to create the request. If cookie information is present (cookies are discussed later in the chapter), it is contained in the Cookie property. Multiple cookies can be more easily accessed by the CookieFields array. If any parameters were passed with the request, they will all be contained in a single string inside the Query property. They will also be broken out into an array in the QueryFields property.

NOTE

When you are passing parameters to a URL, they normally follow a question mark (?) after the URL's name. Multiple parameters are separated by ampersands (&), and if the parameters contain spaces, a plus sign (+) is substituted for the spaces. Therefore, a valid set of parameters might look like this inside an HTML page:

```
<A HREF="http://www.someplace.com/ISAPIApp?Param1=This+
➥Parameter&Param2=That+Parameter">Some Link</A>
```

Most of the information for a TISAPIRequest is revealed in properties, but the class makes public many of the functions used to fill those properties, thus allowing you to access the data directly if you want. TISAPIRequest contains other properties than those discussed here, but these are the main ones you should be interested in. All these properties can be used in your OnAction event handler to determine the type of response your Web server application will provide. If you want to include information about the user's IP address or vary the response based on the type of browser the client is using, you can do that in your OnAction event handler.

You can see what a TISAPIRequest looks like by running the following project in your Web server. Create a new Web server application, bring up the Actions property editor by double-clicking the Actions property in the Object Inspector, and create a new TWebActionItem with

the `PathInfo` set to `http`. Go to the Internet page on the Component Palette and drop a `TPageProducer` (discussed later in this chapter) on the `WebModule`; then add the code shown in Listing 31.2 to the `OnAction` event handler for `/http`.

LISTING 31.2 The `OnAction` Event Handler

```
procedure TWebModule1.WebModule1Actions0Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
   Page: TStringList;
begin
   Page := TStringList.Create;
   try
     with Page do
     begin
          Add('<HTML>');
          Add('<HEAD>');
      Add('<TITLE>Web Server Extensions THTTPRequest Demo</TITLE>');
          Add('</HEAD>');
          Add('<BODY>');
          Add('<H3><FONT="RED">This page displays the properties
      ➥of the HTTP request that asked for it.</FONT></H3>');
Add('<P>');

          Add('Method = ' + Request.Method + '<BR>');
          Add('ProtocolVersion = ' + Request.ProtocolVersion + '<BR>');
          Add('URL = ' + Request.URL + '<BR>');
          Add('Query = ' + Request.Query + '<BR>');
          Add('PathInfo = ' + Request.PathInfo + '<BR>');
          Add('PathTranslated = ' + Request.PathTranslated + '<BR>');
          Add('Authorization = ' + Request.Authorization + '<BR>');
          Add('CacheControl = ' + Request.CacheControl + '<BR>');
          Add('Cookie = ' + Request.Cookie + '<BR>');
          Add('Date = ' + FormatDateTime ('mmm dd, yyyy hh:mm',
              ÂRequest.Date) + '<BR>');
          Add('Accept = ' + Request.Accept + '<BR>');
          Add('From = ' + Request.From + '<BR>');
          Add('Host = ' + Request.Host + '<BR>');
          Add('IfModifiedSince = ' + FormatDateTime ('mmm dd, yyyy hh:mm',
              ÂRequest.IfModifiedSince) + '<BR>');
          Add('Referer = ' + Request.Referer + '<BR>');
          Add('UserAgent = ' + Request.UserAgent + '<BR>');
          Add('ContentEncoding = ' + Request.ContentEncoding + '<BR>');
          Add('ContentType = ' + Request.ContentType + '<BR>');
          Add('ContentLength = ' + IntToStr(Request.ContentLength) + '<BR>');
```

continues

LISTING 31.2 Continued

```
          Add('ContentVersion = ' + Request.ContentVersion + '<BR>');
          Add('Content = ' + Request.Content + '<BR>');
          Add('Connection = ' + Request.Connection + '<BR>');
          Add('DerivedFrom = ' + Request.DerivedFrom + '<BR>');
          Add('Expires = ' + FormatDateTime ('mmm dd, yyyy hh:mm',
               ➥ Request.Expires) + '<BR>');
Add('Title = ' + Request.Title + '<BR>');
          Add('RemoteAddr = ' + Request.RemoteAddr + '<BR>');
          Add('RemoteHost = ' + Request.RemoteHost + '<BR>');
          Add('ScriptName = ' + Request.ScriptName + '<BR>');
          Add('ServerPort = ' + IntToStr(Request.ServerPort) + '<BR>');

          Add('</BODY>');
          Add('</HTML>');
     end;
       PageProducer1.HTMLDoc := Page;
       Response.Content := PageProducer1.Content;
   finally
     Page.Free;
   end;
  Handled := True;
end;
```

Build the project and copy the resulting `Project1.dll` file in the default directory for your ISAPI- or NSAPI-capable Web server. Point your Web browser to `http://<your server>/project1.dll/http`; when you view this application; it will show you all the values of the HTTP fields passed to the server in the request from your browser.

Of course, every request should have a proper response; therefore, Delphi defines the `TISAPIResponse` class to allow you to return information to the requesting client. The most important property of `TISAPIResponse` is the `Content` property. This is the property that will contain the HTML code that is to be displayed for the client.

`TISAPIResponse` contains a number of additional properties that can be set by your application. You can pass version information in the `Version` property. You can tell the client when the information being passed back was last modified with the `LastModified` property. You can pass information about the content, itself, with the `ContentEncoding`, `ContentType`, and `ContentVersion` properties. The `StatusCode` property allows you to return error codes and other status codes to the client.

TIP

Most browsers react in specific ways to certain status codes. You can check the HTTP specification at the Web site `http://www.w3.org` for the specific status codes.

The real power of `TISAPIResponse` comes in its methods. Once you have properly formatted your response, use the `SendResponse` method to force your Web application to send the `TWebResponse` information back to the client. You can send any sort of data back to the client using the `SendStream` method. Also, if your application decides to send the client somewhere other than the response provided by the application itself, it can do so using the `SendRedirect` method. `SendRedirect` is discussed later in the chapter.

Dynamic HTML Pages with HTML Content Producers

Of course, building HTML code dynamically is not the most efficient way to provide Web pages, so Delphi provides a number of tools to make building HTML pages much easier, more efficient, and customizable. `TCustomContentProducer` is an abstract class that provides the basic functionality for handling and manipulating HTML pages. `TPageProducer`, `TDataSetTableProducer`, and `TQueryTableProducer` descend from it. When used together, and with either existing or dynamically created HTML, these classes allow you to create a site based on dynamic HTML pages, including data in tables, hyperlinks, and the full range of HTML capabilities. These controls will not actually create HTML for you, but they make the management of HTML and the dynamic creation of Web pages based on parameters and other inputs quite simple.

TPageProducer

`TPageProducer` is used for the manipulation of straight HTML code. It uses customized HTML tags, replacing them with the proper content. You create, either at design time or runtime, an HTML template that contains special tags that are ignored by standard HTML. The `TPageProducer` can then find these tags and replace them with the appropriate information. The tags can contain parameters for passing information. You can even replace one custom tag with text containing other custom tags, thus allowing you to link page producers together, causing a “daisy chain” effect that enables you to define a dynamic Web page based on differing inputs.

These dynamic tags look just like regular HTML tags, but because they are not standard HTML tags, they are ignored by the client's browser. Such a tag looks like this:

```
<#CustomTag Param1=SomeValue "Param2=Some Value with Spaces">
```

The tag should be surrounded by the less-than (<) and greater-than (>) brackets, and the tag's name must begin with a pound sign (#). The tag name must be a valid Pascal identifier. Parameters with spaces must be entirely surrounded by quotes. These custom tags can be placed anywhere inside your HTML document, even inside other HTML tags.

Delphi provides a number of predefined tag names. None of the values have any special action associated with them; rather, they are defined only for convenience and code clarity. For example, you are not required to use the `tgLink` custom tag for a link, but it makes sense (and is clearer in your HTML templates) if you do so. Note that you can define all your custom tags as you want, and they will all become `tgCustom` values. Table 31.1 shows the predefined tag values.

TABLE 31.1 Predefined Tag Values

Name	*Value*	*Tag Conversion Value*
Custom	`TgCustom`	A user-defined or unidentified tag. It can be converted to any user-defined value.
Link	`TgLink`	This tag should be converted to an anchor value. This is normally a hypertext link or a bookmark value (`<A>..</A>`).
Image	`TgImage`	This tag should be converted to an image tag (`<IMG SRC=...>`).
Table	`TgTable`	This tag should be replaced with an HTML table (`<TABLE>..</TABLE>`).
ImageMap	`TgImageMap`	This tag should be replaced with an image map. An image map defines links based on hot zones within an image (`<MAP>...</MAP>`).
Object	`TgObject`	This tag should be replaced with code that calls an ActiveX control.
Embed	`TgEmbed`	This tag should be converted to a tag that refers to a Netscape-compliant add-in DLL.

Using the `TPageProducer` component is rather straightforward. You can assign HTML code to the component in either the `HTMLDoc` or `HTMLFile` property. Whenever the `Content` property is assigned to another variable (usually the `TISAPIResponse.Content` property), it scans the given HTML, calling the `OnHTMLTag` event whenever a custom tag is found in the HTML. The `OnHTMLTag` event handler looks like this:

```
procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin

end;
```

The `Tag` parameter contains the type of tag found (refer to Table 31.1). The `TagString` parameter holds the value of the whole tag itself. The `TagParams` parameter is an indexed list of each parameter, including the parameter name, the equal sign (=), and the value itself. The `ReplaceText` parameter is a string variable that you will fill with the new value that will replace the tag. The entire tag, including the angle brackets (< and >), is replaced in the HTML code with whatever value is passed back in this parameter.

You can assign an HTML template to the `TPageProducer` in one of two ways. You can create the HTML at runtime as a string and pass it to the `HTMLDoc` property, or you can assign an existing HTML file to the `HTMLFile` property. This allows you to build HTML on the fly or to use existing templates that you have prepared ahead of time.

For example, suppose you have an HTML file called `MYPAGE.HTM` with the following HTML code in it:

```
<HTML>
<HEAD>
    <TITLE>My Cool Homepage</TITLE>
</HEAD>
<BODY>
Howdy <#Name>!  Thanks for stopping by my web site!
</BODY>
</HTML>
```

You can then assign the following code to the `PageProducer.OnHTMLTag` event handler:

```
procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
    case Tag of
       tgCustom: if TagString = 'Name' then ReplaceText := 'Partner';
    end;
end;
```

This results in the following HTML code:

```
<HTML>
<HEAD>
    <TITLE>My Cool Homepage</TITLE>
</HEAD>
<BODY>
```

```
Howdy Partner!  Thanks for stopping by my web site!
</BODY>
</HTML>
```

Suppose that you used this code with the `OnAction` event in a `WebModule`, like this:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
     PageProducer1.HTMLFile := 'MYPAGE.HTM';
     Response.Content := PageProducer1.Content;
end;
```

The newly created page would be sent back to the client when requested. When the `PageProducer.Content` property is called, it makes the given replacement of text for every tag it finds, calling the `OnHTMLTag` event handler for each one. More complex pages might have numerous entries in the `case` statement, replacing various different custom tags with large chunks of HTML, links to other pages, graphics, tables, and so on.

`TCustomPageProducer` objects can also be linked together in a chain. You can use two of them to produce a single page. For example, you might have a basic HTML template that holds standard header and footer code, along with custom tags that define some general values for the page and the location of the main body of the page. You might pass this through one page producer, replacing general data tags with information based on the user's identity. Then, you might replace the main body tag with customized code or more tags based on the information requested by that user. The result could then be passed to yet another `TPageProducer`, which would replace those specific tag values with the appropriate information.

TDataSetTableProducer and TQueryTableProducer

In addition to regular HTML documents, Delphi provides the `TDataSetTableProducer` to allow you to easily and powerfully create HTML tables based on a given dataset. `TDataSetTableProducer` allows you to fully customize all characteristics of the table, within the limits set by HTML. This class can function to a large degree as a `TDBGrid` because you can format individual cells, rows, and columns. You can access data from any dataset available to your system, whether local or remote. This allows you to build enterprise-level Web sites that access data from virtually any source.

`TDataSetTableProducer` behaves a bit differently than the other database controls in that it accesses data directly from a `TDataSet` descendant rather than through a `TDataSource`. It has a `DataSet` property that can be set at design time to any `TDataSet` descendant found in the same `TWebModule`, or at runtime to any dynamically created value. After the `DataSet` property has been set, you can access and configure the `TDataSetTableProducer` to display any of the columns of the given dataset, as desired. The `TableAttributes` property allows you to set the general characteristics of the table, again within the confines of the HTML specification.

The `Header` and `Footer` properties are of type `TStrings` and allow you to add HTML code before and after the table itself. You can use these properties in conjunction with your own dynamically created HTML or with HTML from a `TPageProducer`. For instance, if the main feature of a page is the table, you might use the `Header` and `Footer` properties to fill in the basic structure of the HTML page. If the table is not the main focus of the page, you might choose to use a custom `TTag` in a `TPageProducer` to place the table in the appropriate place. Either way, you can use the `TDataSetTableProducer` to create data-based Web pages.

The `Columns`, `RowAttributes`, and `TableAttributes` properties are where customizing is done for the table to be produced. The `Columns` property hides a very powerful component editor that you can use to set most of the component's attributes.

TIP

Double-click the component itself or the `Columns` property in the Object Inspector to invoke the `Columns` property editor.

The `Caption` and `CaptionAlign` properties determine how the caption of the table will be shown. The `Caption` is the text displayed either above or below the table, serving to explain the table's contents. The `DataSet` property (`Query` in the `TQueryTableProducer`) determines the data to be used in the table.

Other than the way they access data, `TDataSetTableProducer` and `TQueryTableProducer` function identically. They have the same properties and are configured the same way. Because of this, you will create a table that is the result of a simple join and use `TQueryTableProducer` in an example to see how they both work.

Start a new Web application and drop a `TQueryTableProducer` from the Internet page of the Component Palette and a `TQuery` and a `TSession` from the Data Access Palette page onto the `TWebModule`. Set the `QueryTableProducer1.Query` property to `Query1` and the `Query1.DatabaseName` property to `DBDEMOS`. Save the project as `TABLEEX.DPR`. Then set the `Query1.SQL` property as follows:

```
SELECT CUSTNO, ORDERNO, COMPANY, AMOUNTPAID, ITEMSTOTAL FROM CUSTOMER,
➥ORDERS WHERE
   CUSTOMER.CUSTNO = ORDERS.CUSTNO
   AND
   ORDERS.AMOUNTPAID <> ORDERS.ITEMSTOTAL
```

This will produce a small, joined table that has all the customers from the `CUSTOMER.DB` table in the standard `DBDemos` alias who have not yet paid all their orders in full. You can then build a table that shows this data and highlight the amount owed. Set `Query1.Active` to `True` so that the data will be displayed in the `Columns` editor.

NOTE

All Web server applications that will be handling data and using Delphi's data components need to have a `TSession` included in the `WebModule`. Web server applications can be accessed many times concurrently, and Delphi will run each ISAPI or NSAPI server application in a separate thread for each request. As a result, your application needs to have its own, unique session when talking to the BDE. A `TSession with the AutoSessionName property set to True` in your application ensures that each thread has its own session and does not conflict with other threads trying to access the same data. All you need to do is make sure that there is a `TSession` present in your project—Delphi takes care of the rest.

TIP

When you are building Web extension applications, the `TWebApplication.CacheConnections` property can speed up your application. Each time a client makes a request of your ISAPI or NSAPI application, a new thread is spawned to handle your request, in the process creating a new instance of your `TWebModule`. Normally, each thread is executed for a single connection, and the `TWebModule` is destroyed when that connection is closed. If `CacheConnections` is set to `True`, each thread is preserved and reused as needed. New threads are only created when a cached thread is not available. This will speed performance by saving the execution time for creating a `TWebModule` request every time. However, you have to be really careful, because `TWebModule.OnCreate` is called only once for each cached thread. When a cached thread is finished, it remains in the state it was at completion. This might cause problems the next time the thread is used, depending on what happens in your `OnCreate` event. If you depend on `OnCreate` to initialize variables or perform other initialization actions, you might not want to use cached connections. Instead, you should use an additional method that initializes the data for your Web application and then call that in the `BeforeDispatch` event handler. This way, each time a request is made, the data for your Web module will be initialized.

You can check the current number of unused, cached connections with the `TWebApplication.InactiveCount` property. `TWebApplication.ActiveCount` will tell you the number of active connections. These two properties may help you determine a good value for `TWebApplication.MaxConnections`, which limits the total number of connections that the `TWebModule` can handle at once. An exception will be raised if `ActiveCount` ever exceeds `MaxConnections`.

Double-click QueryTableProducer1 to invoke the Columns component editor. In the upper-left area of the component editor, you can set the general properties for the table as a whole. The lower half of the editor contains an HTML control that will display the table as it is currently configured. The upper-right area contains a collection of THTMLTableColumn items that can be configured to determine which fields of the database will be included in the table and how those fields will be displayed. Delphi will automatically import the fields from the TQuery and add them to the fields editor. This application will not display the last field, so select the ItemsTotal field and delete it. In addition, select the AMOUNTPAID field, and set the BgColor property to Lime.

TIP

It might be a good idea to resize the Columns property editor in order to accommodate your table, especially if it will contain a number of columns.

In the upper-left part of the window, set the Border value to 1 so that you will be able to see the border of the table in the component editor as it is built. Set the CellPadding value to 2 to provide a bit of spacing between the border and the text. If you want to add a little color to the table, set the BgColor property to Aqua. This will cause the default background color of the table to be aqua. Note that this is the default color—setting the background color for a row or a column will override this value. In addition, Column color settings take precedence over Row color settings.

When Delphi creates the field columns for the table, it gives the HTML columns headers the names of the fields. However, database field names often do not make nice table column headings, so you can change the default values using the Title property. Title is a compound property, and one of its subproperties is Caption. Set the Title.Caption properties of the four columns to Cust #, Order #, Company, and Amount Owed, respectively. Amount Owed is not quite what the fourth column currently represents, but you will customize the output for this column a little later. The Title property also allows you to customize the vertical and horizontal alignment, as well as the color of the column header cell.

NOTE

TTHMLTableColumn, like other table-related classes, has a Custom property. This property lets you enter a string value for the given item in the table. This value will be entered directly in the HTML tag that defines the given table element. Custom items might include HTML cell, row, or column modifiers not included in the properties of

continues

the class or proprietary HTML extensions. Microsoft Internet Explorer includes a number of table-formatting extensions that allow you to customize the frames of the table. If you want to add these capabilities, make the entry in the `Custom` property in the form of *`paramname=value`*. You can add multiple parameters separated by spaces.

That covers the basic properties for the table that you can set at design time. Now we will discuss the events associated with `TQueryTableProducer` that allow you to customize the table at runtime. `OnCreateContent` occurs prior to any HTML being generated. It contains the `Continue` parameter, a Boolean value that you can set. If your application determines that for some reason the table should not be generated, you can set this parameter to `False`, and no more processing will be done; a call to the `Content` property will return an empty string. It might be used to do such things as prepare the query, set the `TQueryTableProducer.MaxRows` property, or any other processing that you need to do before actually displaying the table.

For instance, in the current example, the application will need to step through each record in the `Query` as the table is drawn. To ensure that as the table is built the query is pointing to the proper record, the application has to manually increment the cursor in the query each time a new row is started. To do that, the query has to start at the beginning, as does the `TQueryTableProducer`. Therefore, a call to `Query1.First` in the `OnCreateContent` event handler ensures that the query and the HTML table are in sync with each other. Therefore, add the following code to the event handler for `QueryTableProducer1.OnCreateContent`:

```
procedure TWebModule1.QueryTableProducer1CreateContent(Sender: TObject;
  var Continue: Boolean);
begin
    QueryTableProducer1.MaxRows := Query1.RecordCount;
    Query1.First;
    Continue := True;
end;
```

The `OnGetTableCaption` event allows you to format the table's caption however you want. Double-clicking the event in the Object Inspector yields this event handler:

```
procedure TWebModule1.QueryTableProducer1GetTableCaption(Sender: TObject;
  var Caption: String; var Alignment: THTMLCaptionAlignment);
begin

end;
```

The `Caption` parameter is a variable parameter that will hold the end result of your caption. You can manipulate this parameter as you please, including adding HTML tags to size, color, and format the font of the table's caption. You can use the `Alignment` parameter to determine whether the caption is aligned at the top or the bottom of the table.

Create an `OnGetTableCaption` for the example that you have been working on by double-clicking it in the Object Inspector. Enter the following code to format the table's `Caption` in order to make it stand out a bit more on the page (this change will not be reflected on the HTML table shown in the `Columns` property editor):

```
procedure TWebModule1.QueryTableProducer1GetTableCaption(Sender: TObject;
  var Caption: String; var Alignment: THTMLCaptionAlignment);
begin
  Caption := '<B><FONT SIZE="+2" COLOR="RED">Delinquent Accounts</FONT></B>';
  Alignment := caTop;
end;
```

The `OnFormatCell` event can be used to change the appearance of an individual cell. In this example, you can add code to highlight the `Amount Owed` cell of any company that has not paid its bill in full. This gets a little trickier than with the regular grids, because `TQueryTableProducer` only provides you with string values. However, as mentioned earlier, you can use the `CellRow` and `CellColumn` parameters to move the cursor of the `TQuery` along as the table is built, gathering the proper data and making calculations as each row is processed.

The `OnFormatCell` event handler passes you the information about the current cell being formatted in the `CellRow` and `CellColumn` parameters. These are both zero-based. The rest of the parameters are variable parameters to which you can assign values, depending on your application's logic. You can adjust the horizontal and vertical alignment of the data in the cell with the `Align` and `VAlign` parameters. You can pass additional `Custom` parameters for the cell in the `CustomAttrs` parameter, and of course, you can alter the actual text of the cell with the `CellData` parameter.

The `CellData` parameter is of type `string`, which limits your ability to process it in its native format. If the data were actually stored in the database as an integer, you would have to call `StrtoInt` to convert it back to a usable number. The following code illustrates how you might gather the actual `TField` values for the given cell. Perhaps future versions of Delphi will pass the `TField` value into the `OnFormatCell` event handler in addition to or in place of the string value. Add the code in Listing 31.3 to the `OnFormatCell` event handler for `TQueryTableProducer`.

LISTING 31.3 The `OnFormatCell` Event Handler

```
procedure TWebModule1.QueryTableProducer1FormatCell(Sender: TObject;
  CellRow, CellColumn: Integer; var BgColor: THTMLBgColor;
  var Align: THTMLAlign; var VAlign: THTMLVAlign; var CustomAttrs,
  CellData: String);
```

continues

LISTING 31.3 Continued

```
  Owed, Paid, Total: Currency;
begin
     if CellRow = 0 then Exit; // Don't process the header row
     if CellColumn = 3 then //if the column is the Amount Owed Column
     begin
         //Calculate the amount that the company owes
         Paid := Query1.FieldByName('AmountPaid').AsCurrency;
         Total := Query1.FieldByName('ItemsTotal').AsCurrency;
         Owed := Total - Paid;
         //Set CellData to amount owed
         CellData := FormatFloat('$0.00', Owed);
         //if it is greater than zero, then highlight the cell.
         if Owed > 0 then
         begin
            BgColor := 'RED';
         end;
         Query1.Next; //Advance the query since we came to the end of a row
     end;
end;
```

This code gathers the data for each unpaid account, subtracts the Amount Owed from the Amount paid, and then highlights in red the accounts that owe money. It illustrates how you can use the current cursor of the `TQuery` component to access the data being displayed in the HTML table.

Next, add the following strings to the `TQueryTableProducer.Header` property:

```
<HTML>
<HEAD>
    <TITLE>Delinquent Accounts</TITLE>
</HEAD>
<BODY>
<CENTER><H2>Big Shot Widgets</H2></CENTER>
<P>
The Accounts highlighted in red are late in paying:
<P>
```

Now add this to the `TQueryTableProducer.Footer` property:

```
<P>
<I>This information is to be kept in the strictest confidence</I><P>
<B><I>Copyright 1999 by BigShotWidgets</I></B><P>
</BODY>
</HTML>
```

This will cause the table to be placed between these two sets of HTML code, thus causing a complete page to be created when the `Content` property of `TQueryTableProducer` is called in the following code.

Finally, go back to the main `TWebModule` of your application and add a single `Action`, setting its `PathInfo` to `/TestTable`. In its `OnAction` event handler, add the following code:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
     Response.Content := QueryTableProducer1.Content;
end;
```

Then compile the project and make sure that the resulting DLL is accessible by your Web server. Now, if you call the URL `http://`*`<your server>`*`/tableex.dll/TestTable`, you will see the table with the header and footer text as well as the positive amounts owed highlighted in red, as shown in Figure 31.2.

Delinquent Accounts - Microsoft Internet Explorer

http://revelation/scripts/tableex.dll/testtable

Big Shot Widgets

The Accounts highlighted in red are late in paying:

Delinquent Accounts

Customer Number	Order #	Company Name	Amount Owed
1351	1003	Sight Diver	$1250.00
1510	1008	Ocean Paradise	$1449.50
1645	1014	Action Club	$0.00
1984	1017	Adventure Undersea	$10195.00
2135	1019	Frank's Divers Supply	$20602.00
2156	1095	Davy Jones' Locker	$7531.75
3615	1097	Marina SCUBA Center	$12953.60
1984	1099	Adventure Undersea	$859.95
2165	1101	Shangri-La Sports Center	$11629.85
4531	1112	On-Target SCUBA	$5565.00
4684	1115	Underwater Fantasy	$4894.95

FIGURE 31.2
A table-based Web page.

Maintaining State with Cookies

The HTTP protocol is a powerful tool, but one of its weaknesses is that it is *stateless*. This means that after an HTTP conversation has been completed, neither the client nor the server

has any memory at all that the conversation even took place, much less what it was about. This can present a number of problems for applications that run across the Web, because the server is not able to remember important items such as passwords, data, record positions, and so on that have been sent to the client. Database applications are particularly affected as they often rely on the client knowing which record is the current record back on the server.

The HTTP protocol provides a basic method for writing information on the client's machine to allow the server to get information about the client from previous HTTP exchanges. Called by the curious name *cookies*, they allow the server to write state information into a file on the client's hard drive and to recall that information at a subsequent HTTP request. This greatly increases a server's capabilities with respect to dynamic Web pages.

Cookies are no more than text values in the form of *CookieName=CookieValue*. A cookie should not include semicolons or commas. The user can refuse to accept cookies, so no application should ever assume that a cookie will be present. Cookies are becoming more and more prevalent as Web sites get more and more sophisticated. If you are a Netscape user, you might be surprised by what you find in your COOKIES.TXT file. Internet Explorer users might peek into the \WINDOWS\COOKIES folder. If you want to track cookies as they are set on your machine, both of these browsers allow you to approve individual cookie settings within their security preference settings.

Managing cookies in Delphi is, pardon the pun, a piece of cake. The THTTPRequest and THTTPResponse classes encapsulate the handling of cookies quite cleanly, allowing you to easily control how cookie values are set on a client's machine as well as to read what cookies have been previously set.

The work of setting a cookie is all done in the TWebResponse.SetCookieField method. Here, you can pass a TStrings descendant full of cookie values, along with the restrictions placed on the cookies.

The SetCookieField method is declared as follows in the HTTPAPP unit:

```
procedure SetCookieField(Values: TStrings; const ADomain, APath: string;
➥AExpires: TDateTime; ASecure: Boolean);
```

The Values parameter is a TStrings descendant (you will probably use a TStringList) that holds the actual string values of the cookies. You can pass multiple cookies in the Values parameter.

The ADomain parameter allows you to define in which domain the given cookies are relevant. If no domain value is passed, the cookie will be passed to every server to which a client makes a request. Normally, a Web application will set its own domain here so that only the pertinent cookies are returned. The client will examine the existing cookie values and return those cookies that match the given criteria.

For example, if you pass `widgets.com` in the `ADomain` parameter, all future requests to a server in the `widgets.com` domain will pass along the cookie value set with that domain value. The cookie value will not be passed to other domains. If the client requests `big.widgets.com` or `small.widgets.com`, the cookie will be passed. Only hosts within the domain can set cookie values for that domain, thus avoiding all sorts of potential for mischief.

The `APath` parameter allows you to set a subset of URLs within the domain where the cookie is valid. The `APath` parameter is a subset of the `ADomain` parameter. If the server domain matches the `ADomain` parameter, the `APath` parameter is checked against the current path information of the requested domain. If the `APath` parameter matches the pathname information in the client request, the cookie is considered valid.

For example, following the preceding example, if `APath` contained the value `/nuts`, the cookie would be valid for a request to `widgets.com/nuts` and any further paths, such as `widgets.com/nuts/andbolts`.

The `AExpires` parameter determines how long a cookie should remain valid. You can pass any `TDateTime` value in this parameter. Because the client could be anywhere in the world, this value needs to be based on the GMT time zone. If you want a cookie to be valid for 10 days, pass `Now + 10` as a value.

If you want to delete a cookie, pass a date value that is in the past (that is, a negative value) and that will invalidate the cookie. Note that a cookie may become invalid and not be passed, but that does not necessarily mean that the cookie is actually removed from the client's machine.

The final parameter, `ASecure`, is a Boolean value that determines whether the cookie can be passed over nonsecure channels. A `True` value means that the cookie can only be passed over the HTTP-Secure protocol or a Secure Sockets Layer network. For normal use, this parameter should be set to `False`.

Your Web server application receives cookies sent by the client in the `TWebRequest.CookieFields` property. This parameter is a `TStrings` descendant that holds the values in an indexed array. The strings are the complete cookie value in *`param=value`* form. They can be accessed like any other `TStrings` value. The cookies are also passed as a single string in the `TWebRequest.Cookie` property, but normally you would not want to manipulate them here. You can assign the cookies directly to an existing `TStrings` object with the `TWebRequest.ExtractCookieFields` method.

A simple example can illustrate the ease with which Delphi deals with cookies. First, create a new Web Application and add the `WebUtils` unit to your `uses` clause. The `WebUtils` unit is included on the CD-ROM accompanying this book. Then create a new Web server application

and give it two actions—one named `SetCookie` and the other `GetCookie`. Set the code in the `OnAction` event for `SetCookie` to the following:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  List: TStringList;
begin
  List := TStringList.Create;
  try
     List.Add('LastVisit=' + FormatDateTime('mm/dd/yyyy hh:mm:ss', Now));
     Response.SetCookieField(List, '', '', Now + 10, False);
     Response.Content := 'Cookie set -- ' + Response.Cookies[0].Name;
  finally
    List.Free;
  end;
  Handled := True;
end;
```

The `OnAction` code for `GetCookie` should be as follows:

```
procedure TWebModule1.WebModule1WebActionItem2Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  Params: TParamsList;
begin
     Params := TParamsList.Create;
     try
       Params.AddParameters(Request.CookieFields);
       Response.Content := 'You last set the cookie on ' + Params['LastVisit'];
     finally
       Params.Free;
     end;
end;
```

Set up a Web page that calls the following two URLs:

```
http://<your server>/project1.dll/SetCookie
http://<your server>/project1.dll/GetCookie
```

NOTE

The `TParamsList` class is part of the `WebUtils` unit included on the CD-ROM. It is a class that automatically parses out parameters from a `TStrings` descendant and allows you to index them by the parameter's name. For instance, `TWebResponse` gathers all the cookies passed in an HTTP response and places them in the `CookieFields`

property, which is a `TStrings` descendant. The cookies are in the form *CookieName=CookieValue*. `TParamsList` takes these values, parses them, and indexes them by the parameter name. Therefore, the preceding parameter could be accessed with `MyParams['CookieName']`, which would return *CookieValue*. You can use this class, or you can use the `Values` property found in the `TStrings` class included in the VCL.

Set the cookie by calling for the first URL from a Web page in the same directory as the DLL. This will set a cookie on the client machine that lasts for 10 days and contains the date and time that the request was made in a cookie called `LastVisit`. If you have your Web browser set to accept cookies, it should ask you to confirm the writing of the cookie. Then call the `GetCookie` action to read the cookie, and you should see the date and time that you last called the `SetCookie` action.

Cookies can contain any information that can be stored in a string. Cookies can be as big as 4KB, and a client can store as many as 300 cookies. Any individual server or domain is limited to 20 cookies. Cookies are powerful but, as you can see, you should try to limit their use. They certainly cannot be used to store large amounts of data on a client's machine.

Very often, you will want to store more information about a user than can be stored in a cookie. Sometimes you will want to keep track of a user's preferences, address, personal information, or even items in a "grocery cart" that are to be purchased from your e-commerce site. This information can easily become rather voluminous. Rather than try to store all this information in the cookie itself, it is often better to encode user information into a cookie rather than storing the information as is. For instance, in order to store a collection of user preferences that are really Boolean values, you might store them in binary format inside the cookie. Therefore, a cookie value of `'1001'` might mean that the user does want further email updates, does not want his or her email address given to other users, does not want to be added to your list server, and does want to join your online discussion groups. You can use characters or numbers to further encode even more data about a user in a cookie.

You can also store a user identification value in a cookie that uniquely identifies a user. You can then retrieve that value from the cookie and use it to look up the user's data in a database. That way, you would be able to minimize the amount of data stored on the user's computer and maximize your control over the information that you maintain about a user.

Cookies can provide a powerful and easy way to maintain data about your users between individual HTTP sessions.

Redirecting to a Different Web Site

Often, a given URL may want to change the destination of a user's request. A Web application may want to process some data based on a request and then serve back a page that may vary depending on the nature of the request or a database entry. Web advertising does this frequently. Often an ad graphic will point to another URL within the domain where it appears, but clicking it takes the user to the advertiser's home page. Along the way, data is gathered about the request and then the client is handed off to the advertiser's page. Frequently, the HTML code for the advertisement's graphic will contain parameters that describe the ad to the server. The server can log that information and then pass the client on to the proper page. This technique is called *redirection*, and it can be very useful for a number of tasks.

Delphi's `TWebResponse` class includes a method called `SendRedirect`. It takes a single string as a parameter that should contain the full address of the site to which the client should be redirected. The method is declared as follows:

```
procedure SendRedirect(const URI: string); virtual; abstract;
```

`SendRedirect` is declared as an abstract method in `HTTPAPP.PAS` and defined in `ISAPIAPP.PAS`.

A Web server could easily process an HTTP request that includes parameters and then pass that request to a site named by one of those parameters. For instance, if a cool GIF file is on a page, and the whole graphic is wrapped up as a hyperlink, the URL assigned to it might look something like this:

```
<A
HREF="http://www.somecoolplace.com/transfer?www.borland.com&coolgif.gif&borland">
➥<IMG SRC="coolgif.gif"></A>
```

Given that information, an `OnAction` event in a Web server application named `/transfer` might resemble the following code fragment:

```
procedure TWebModule1.WebModule1WebActionItem3Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
 {Process Request.QueryFields[1] perhaps placing it in a database.
  It holds the name of the GIF file that caused the user to click on it.
  You might want to track the GIFs that are the most effective.
  Then you can keep track of how many hits a particular company is
  getting from your site by tracking the company name that is getting
  requested in the Request.QueryFields[2] parameter}
  //Then, you can call this to send the user on his merry way...
  Response.SendRedirect(Request.QueryFields[0]);
end;
```

By using this technique, you can create a generic transfer application that processes every advertisement on a site. Of course, there may be other reasons for calling `SendRedirect` than just advertising. You can use `SendRedirect` whenever you want to keep track of specific URL requests and any data that might be associated with a particular hyperlink. Simply gather the data from the `QueryFields` property and then call `SendRedirect` as needed.

Retrieving Information from HTML Forms

HTML-based forms are growing in use with the growth of the Internet and intranets. It is not a surprise that Delphi makes gathering information from forms easy. This chapter does not cover the details of creating an HTML-based form and the controls that go with it, but rather it deals with how Delphi handles the forms and their data.

On the CD-ROM in the back of this book is a straightforward guest book application that gathers the input from an HTML form and makes entries into a database table. Opening the `INDEX.HTM` file in your browser can access the application. The HTML form for the guest book, `GUEST.HTM`, uses the following line to define the form and the action to take when the user clicks the Submit button:

```
<form method="post" action="guestbk.dll/form">
```

This code causes the form to "post" its data when asked to do so and to call the given DLL `OnAction` event. The form allows the user to enter his or her name, email address, home town, and comments. When the user clicks the Submit button, that information is gathered up and passed to the Web application.

The action with the name `/form` then receives the data in the `Request.ContentFields`, in the form of standard HTTP parameters. `ContentFields` is a `TStrings` descendant that holds the contents of the submitted form. The application contains a `TTable` named `GBTable` that is referenced by the `GBDATA` alias. You will need to create this alias and point it to the `/GBDATA` directory where the Paradox tables reside in order to run the guest book. Listing 31.4 shows the code that receives the content of the form and enters it into the database.

LISTING 31.4 Code for Retrieving Content of a Form

```
var
   MyPage: TStringList;
   ParamsList: TParamsList;
begin
     begin
          ParamsList := TParamsList.Create;
          try try
            ParamsList.AddParameters(Request.ContentFields);
```

continues

LISTING 31.4 Continued

```
            GBTable.Open;
            GBTable.Append;
            GBTable.FieldByName('Name').Value := ParamsList['fullnameText'];
            GBTable.FieldByName('EMail').Value := ParamsList['emailText'];
            GBTable.FieldByName('WhereFrom').Value :=
            ➥ParamsList['wherefromText'];
GBTable.FieldByName('Comments').Value := ParamsList['commentsTextArea'];
            GBTable.FieldByName('FirstTime').Value :=
            ➥(CompareStr(ParamsList['firstVisitCheck'], 'on') = 0);
            GBTable.FieldByName('DateTime').Value := Now;
            GBTable.Post;
          except
              Response.Content := 'An Error occurred in processing your data.';
               Handled := True;
          end;
          finally
            ParamsList.Free;
            GBTable.Close;
          end;
     end;
```

The code first inserts the `ContentFields` property into a `TParamsList`. It then opens the `GBTable` and inserts the data from the form into the appropriate fields. The code in Listing 31.4 is relatively straightforward.

The next portion of the code, shown in Listing 31.5, creates an HTML response that thanks the user for making an entry. It uses some of the data from the form to address the user by name, and it also confirms the user's email address.

LISTING 31.5 Code for Creating an HTML Response

```
MyPage := TStringList.Create;
  ParamsList := TParamsList.Create;
     try
       with MyPage do
       begin
         Add('<HTML>');
         Add('<HEAD><TITLE>Guest Book Demo Page</TITLE></HEAD>');
         Add('<BODY>');
         Add('<H2>Delphi Guest Book Demo</H2><HR>');
         ParamsList.AddParameters(Request.ContentFields);
         Add('<H3>Hello <FONT COLOR="RED">'+ ParamsList['fullnameText']
          ➥+'</FONT> from '+ParamsList['wherefromText']+'!</H3><P>');
        Add('Thanks for visiting my homepage and making
```

```
      ➥an entry into my Guestbook.<P>');
      Add('If we need to e-mail you, we will use this address -- <B>'
        ➥+ParamsList['emailText']+'</B>');
      Add('<HR></BODY>');
      Add('</HTML>');
    end;
   PageProducer1.HtmlDoc := MyPage;
  finally
    MyPage.Free;
    ParamsList.Free;
  end;
  Response.Content := PageProducer1.Content;
  Handled := True;
```

Finally, the application provides a summary of all guest book entries in the `/entries` action.

Data Streaming

Most of the data you will be providing to clients by HTTP requests will probably be HTML-based pages. However, there may be a time when you want to send other types of data in response to a user's request. Sometimes you might want to provide different graphics or sounds based upon a user's input. You may have a special data format that you want to send down the pipe to a user that can be specially handled by the client's browser. For instance, Netscape provides a plug-in architecture that allows developers to write extensions to the Navigator browser to handle any type of data. RealAudio, Shockwave, and other types of data streaming are examples of Netscape plug-ins that can extend the power of the client's browser.

Whatever the type of data you want to transmit, Delphi makes it easy to stream data back to a client. The `TWebResponse.SendStream` method along with the `TWebResponse.ContentStream` property enable you to send any type of data back to the client by loading it into a Delphi stream class. Of course, you will need to let the client's browser know what type of data is being sent, so you will need to set the `TWebResponse.ContentType` property as well. Setting this string value to an appropriate MIME type will allow the browser to properly handle the incoming data. For instance, if you want to stream a Windows WAV file, you would set the `ContentType` property to `'audio/wav'`.

NOTE

MIME stands for *Multipurpose Internet Mail Extensions*. MIME extensions were developed to allow clients and servers to pass data by email that was more complex than the standard text passed in most emails. Browsers and the HTTP protocol have

continues

adapted MIME extensions to allow you to pass almost any sort of data from a Web server to a Web browser. Your Web browser contains a rather large list of these MIME types, and it associates a particular application or plug-in with each MIME type. When the browser gets that type, it looks up which application should be used to handle that particular MIME type and passes the data to it.

Using streams allows you to pass any type of data from virtually any source on your Web server's machine. You can pass data from files that reside on your server or anywhere on your network, from Windows resources built into your ISAPI DLL or other DLLs available to your ISAPI DLL, or you can even construct the data on the fly and send it to the client. There is really no limit to how or what you can send, as long as your client's browser knows what to do with the data.

Now we will construct a simple Web application that illustrates what can be done. You will set up a Web page that displays images from various sources. The application will process the image data as needed and return it to the client as requested. This will be surprisingly easy, because Delphi provides numerous different stream classes that make gathering data into a stream very easy, and the ISAPI extension classes make sending that data a snap as well.

To build the data streaming example, select File, New from the main menu and choose Web Server Application from the resulting dialog. This will give you a `TWebModule`. Go to the Web module, select it, and then go to the Object Inspecto. Double-click the `Actions` property and create three actions called `/file`, `/bitmap`, and `/resource`.

Select the `/file` action, go to the Object Inspector, and select the Events page. Create an `OnAction` event and then add the following code to the event handler:

```
procedure TWebModule1.WebModule1WebActionItem2Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  FS: TFileStream;
begin
    FS := TFileStream.Create(JPEGFilename, fmOpenRead);
    try
      Response.ContentType := 'image/jpeg';
      Response.ContentStream := FS;
      Response.SendResponse;
      Handled := True;
    finally
      FS.Free;
    end;
end;
```

The preceding code is pretty straightforward. If you set up the code from the CD-ROM on your computer as described earlier, there should be a JPEG file called `TESTIMG.JPG` in the `\bin` directory. The `OnAction` event handler creates a `TFileStream` that loads that file. It then sets the proper MIME type to tell the client browser that a JPEG file is coming, and it then assigns the `TFileStream` to the `Response.ContentStream` property. The data is then returned to the client by calling the `Response.SendResponse` method. As a result, in the accompanying HTML file, there should be a picture of a rose on the provided HTML page.

NOTE

In the HTML that displays this JPEG file in your browser, you can simply place the reference to the Web application's `Action` property directly in the `IMG` tag, like so:

```
<IMG SRC="../bin/streamex.dll/file" BORDER=0>
```

The streaming examples can be displayed by means of the `INDEX.HTM` page in the `\STREAMS` directory

The application is able to find the JPEG file because when it was created, it set the `JPEGFilename` variable as follows:

```
procedure TWebModule1.WebModule1Create(Sender: TObject);
var
  Path: array[0..MAX_PATH - 1] of Char;
  PathStr: string;
begin
  SetString(PathStr, Path, GetModuleFileName(HInstance, Path, SizeOf(Path)));
  JPEGFilename := ExtractFilePath(PathStr) + 'TESTIMG.JPG';
end;
```

The `/bitmap` action will load a different image, but in a totally different way. The code for this action is a bit more complicated. It looks like this:

```
procedure TWebModule1.WebModule1WebActionItem3Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  BM: TBitmap;
  JPEGImage: TJPEGImage;
begin
    BM := TBitmap.Create;
    JPEGImage := TJPEGImage.Create;
    try
      BM.Handle := LoadBitmap(hInstance, 'ATHENA');
      JPEGImage.Assign(BM);
      Response.ContentStream := TMemoryStream.Create;
```

```
      JPEGImage.SaveToStream(Response.ContentStream);
      Response.ContentStream.Position := 0;
      Response.SendResponse;
      Handled := True;
    finally
      BM.Free;
      JPEGImage.Free;
    end;
end;
```

It takes a bit more work to get a bitmap converted to a JPEG and streamed out to the client. A `TBitmap` is used to grab the bitmap out of the resource file. A `TJPEGImage` from the `JPEG` unit is created and will convert the bitmap to a JPEG file.

The `TBitmap` class is created and then the Windows API call `LoadBitmap` is used to grab the bitmap from the resource named `'ATHENA'`. `LoadBitmap` returns the bitmap's handle, which is assigned to the `Handle` property. The bitmap itself then is assigned immediately to the `TJPEGImage`. The `Assign` method is overloaded and contains the smarts to convert the bitmap to a JPEG.

Next comes a nice example of polymorphism. `Response.ContentStream` is declared as a `TStream`, an abstract class. Because of the power of polymorphism, you can create it as any type of `TStream` descendant you like. In this case, it is created as a `TMemoryStream` and used to hold the JPEG with the `TJPEGImage.SaveToStream` method. Now the JPEG is in a stream and can be sent out. An important but easy-to-forget step is to return the position of the stream to zero after saving the JPEG into it. If this is not done, the stream will be positioned at the end, and no data will be streamed out to the client. After all that is completed, the `Response.SendResponse` method is called to send out the data stored in the stream. The result in this case is the bust of Athena from Delphi's About box.

Another way to load a JPEG is by using a resource entry. You can load a JPEG into an RES file using the following code in an RC file and then compiling it using `BRCC32.EXE`. If you load it as `RCDATA`, you can use the `TResourceStream` class to easily load it and send it to the client browser. `TResourceStream` is a very powerful class that will load a resource from either the EXE file itself or a resource located in an external DLL file. The `/resource` action illustrates how to do this by loading the JPEG from the resource named `'JPEG'` that is compiled into the EXE:

```
procedure TWebModule1.WebModule1WebActionItem4Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.ContentStream := TResourceStream.Create(hInstance,
   ➥'JPEG', RT_RCDATA);
  Response.ContentType := 'image/jpeg';
```

```
  Response.SendResponse;
  Handled := True;
end;
```

This code sends the data to the client a little differently. It is much more straightforward and is again a nice example of polymorphism in action. A `TResourceStream` is created and assigned to the `ContentStream` property. Because the `TResourceStream`'s constructor loads the resource into the stream, no further action needs to be taken on the stream, and a simple call to `Response.SendResponse` sends the data down the stream.

The final example streams out a WAV file that is stored as an `RCDATA` resource. This example uses the `Response.SendStream` method to send out a stream created within the method. This illustrates yet another way of sending stream data. You can create a stream, manipulate and modify it as needed, and send it directly back to the client with the `SendStream` method. This action should cause your browser to play a WAV file of a dog barking. Here is the code:

```
 procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  RS: TResourceStream;
begin
    RS := TResourceStream.Create(hInstance, 'BARK', RT_RCDATA);
    try
      Response.ContentType := 'audio/wav';
      Response.SendStream(RS);
      Handled := True;
    finally
      RS.Free;
    end;
end;
```

Summary

This chapter shows you how to build Web server extensions using the ISAPI/NSAPI extensions. This information is easily transferable to the CGI applications that Delphi produces. We discussed the HTTP protocol and how Delphi encapsulates it in its `TWebRequest` and `TWebResponse` classes. We showed you how to build applications using the `TWebModule` and its `OnAction` events with dynamic HTML. We then illustrated custom HTML documents with the `TContentPageProducer` descendants. We also discussed accessing data and building HTML tables using the `TQueryTableProducer`. Then, we discussed how to handle cookies and the content of HTML forms. Finally, we showed you how to stream custom content back to the client. In the next chapter, "MIDAS Development," we will get back to a database-centric way of thinking as you learn about the MIDAS multitier technology.