4

Extending HTML with iHTML

WebSite Pro contains iHTML 2.1 Professional, the Inline HTML server extension toolkit. The iHTML extension to HTML is more than a simple scripting language. Using iHTML 2.1, you can enhance your web site with order query or order entry systems, online content maintenance, random advertising, site changes based on time of day or user, database-generated sites that appear static, dynamic VRML, and more.

iHTML 2.1 is implemented as a dynamic linked library (DLL). You place iHTML tags directly into regular HTML files. These tags allow you to do such things as access databases; perform math calculations; and define IF/THEN constructs, loops, and error handlers. By combining iHTML with Java Script within the same page, you can build some powerful combinations of server and client-side programming. The iHTML extension specifically addresses the following server-side needs:

- Database access. There are many products available for web-to-database connectivity. What sets iHTML apart from the rest is its ability to access multiple data sources on one HTML page. iHTML also has numerous database cataloging capabilities, can handle multiple concurrent connections, and can perform database joins between different database systems using ODBC.
- Flow control and error handling. iHTML adds logical processing to web pages with WHILE loops, IF/ELSE constructs, CASE statements and FOR/NEXT looping capabilities. iHTML allows you to include one HTML file within another.

Furthermore, when the iHTML engine has finished processing one page, it can direct the browser to another page with iHTML.

- Dynamic graphics. With iHTML's powerful graphics engine, you can program graphics, draw images on the fly, or overlay graphic images within an HTML document. By combining dynamic graphics with database access, you can generate charts and graphs from data stored in a corporate database system. All processing is done on the server and no plug-ins or special browsers are required. iHTML can also natively load more than 10 different graphics file formats, such as BMP, JPEG, PCX, and TIF and do the conversion to noncompressed GIF and XBM on the fly.
- Background page processing. iHTML's unique background page processing feature runs a page in the background, performing specified functions. For example, by combining background page processing with iHTML's Internet features, you can build a mailing list manager that polls a mail server, queries a database for subscriber email addresses, and redistributes that mail to the subscribers.
- Internet features. Many scripting languages available today are ports of standard languages to the web. They do not address the specific features available on the Internet, such as DNS lookup, POP and SMTP mail, web advertisement management, server push technology, and so on. iHTML is designed for the Internet, giving you tools for creating ambitious web-based systems.
- Traditional programming functions. Like all good programming languages, iHTML includes features to manipulate strings, do mathematical calculations, and use or create dates. iHTML can create variables, find lengths of strings, and perform most of the functions you would find in a traditional language.

Note

In the iHTML Enterprise edition, iHTML also has tags that perform hardware manipulation including access to joysticks, serial ports, pagers, and fax machines. It is possible to use iHTML tags to make Registry setting changes within Windows NT. In fact, iHTML can configure itself with the back page processor application.

If you know HTML, it is easy to learn iHTML because the language has been kept as close to the HTML standard tag structure as possible. This chapter tells you what you need to know to begin using iHTML. You'll find instructions for setting up your data sources, an overview of how iHTML processes iHTML files, and a simplified introduction to the scripting language followed by the details that make everything work correctly. There are several examples to show you exactly how to accomplish common tasks and then begin to explore some of the advanced features, so that you can use iHTML to its fullest.

Installing iHTML

To install iHTML, insert the WebSite Pro CD into your CD drive and click iHTML on the opening screen. This launches the iHTML Installer Wizard, which installs iHTML and various test files. The wizard sets the data source for the test files as the default data source. You can change the default with the Enterprise Manager, described later in this chapter in "Configuring iHTML with the Enterprise Manager." For those who have no ODBC driver on their system, you'll find some ODBC 3.5 drivers and an installation application in the ODBC directory on the CD.

Installing ODBC Drivers

iHTML supports a number of database formats. To enable iHTML to exchange data within these databases, you must first install their client drivers on the web server. For a complete list of database formats that are supported by iHTML 2.1, refer to "Supported ODBC Database Drivers," later in this chapter.

iHTML uses the full capabilities of the most common Database Management Systems (DBMSs), including SQL Server, FoxPro, Oracle, and Sybase. Therefore, you must install the ODBC drivers required for your specific database system before attempting to refer to them in iHTML. For iHTML, the ODBC driver must be set up as a system Data Source Name (DSN). These ODBC drivers are provided with your database system kit (some are also included with Windows).

The following steps demonstrate the installation of the Microsoft Access drivers.

1. Insert the ODBC diskette (in the Microsoft Access installation kit) into your floppy drive.
2. From the Start menu, choose Run. At the prompt, type `A:\SETUP` (or simply run it from the Windows Explorer).
3. Select the ODBC drivers for the databases you require access to (checkbox).
4. Select the Install ODBC Administration Utility checkbox. Click Continue. This creates the *adm.* program under *C:\ODBC*. You can change this path if needed.
5. From the list of available drivers, select the database you want to connect to.
6. Select the database servers on the network where you want to connect. For example, select MS Access from Installed Drivers and then add a new name (e.g., enter a name for the MS Access server such as MSAccess1).
7. More information will then be displayed for MS Access. Click OK to continue.
8. Restart Windows.

9. System Data Source drivers (DSN) are required to setup on Windows NT. To install these, access the Control Panel and click the 32-Bit ODBC icon. and then click the system DSN button. Click Add and then select the Microsoft Access Driver (**.mdb*) from the list of installed drivers.
10. Click OK and the Setup dialog box appears.
11. Type in a name to refer to this data source and add a description, if you want.
12. Click Select to access a directory dialog. Select the Microsoft Access database that you want this ODBC driver to talk to and click OK. The data source is now set up for use with iHTML. This completes the installation procedure for Microsoft Access. A similar procedure would be used for your other database ODBC drivers.

Supported ODBC Database Drivers

The following is a list of Database Management Systems (DBMSs) which are ODBC compliant and therefore supported by iHTML 2.1. If your database is ODBC compliant, there are drivers for iHTML 2.1 to use to make queries to the database. Any database commands such as stored procedures native to the data source will also operate properly.

- Borland Paradox
- Borland dBase
- Microsoft Access
- Microsoft SQL Server
- Microsoft Excel
- FoxPro
- Oracle
- Sybase
- Progress
- Ingres
- IBM DB2
- Informix
- Basis International (BBX)

Understanding iHTML

To effectively utilize the full capabilities of iHTML, you need to understand how it works with WebSite Pro and how iHTML parses the tags and variables that carry your instructions. This section describes how iHTML pages are handled by the server. It also introduces the tags and variables and provides the grammar rules iHTML uses in parsing a page.

How WebSite Pro and iHTML Work Together

The iHTML grammar is interpreted by the server, where the iHTML tags and variables are detected and resolved into data. The resolved data may contain still more iHTML extensions. When the document arrives at the browser, it is HTML compliant. This eliminates the need for anything other than an HTML-compliant browser to access the features of iHTML (no plug-ins or special browsers are needed). All web users experience the full functionality of your web pages. In fact, iHTML 2.1 is not only an HTML extension; it works with any HTTP-served data. For example, you could work with Virtual Reality Markup Language (VRML) to create a 3D web site.

When WebSite Pro receives a request for an HTML page, it checks to see if the `<!ihtml>` tag is on the page. If it is, the iHTML engine needs to parse the page. The iHTML engine checks for tags that start with `<i` and, if it recognizes the tag, resolves the tag to data. The data may be from a local or a remote system. iHTML is recursive, so there may be variables and other tags within variables or a database. When these are resolved, the new results on the page are resolved. In this way, your HTML pages can change on the fly. The iHTML tag resolution process continues until the document contains only standard HTML tags. (In some cases, you must specifically request recursive processing; see “Recursive Processing” later in this chapter.)

Once all processing is done, WebSite Pro sends the page to the client browser. Figure 4-1 shows the entire process of interpreting an iHTML page and sending it to the browser.

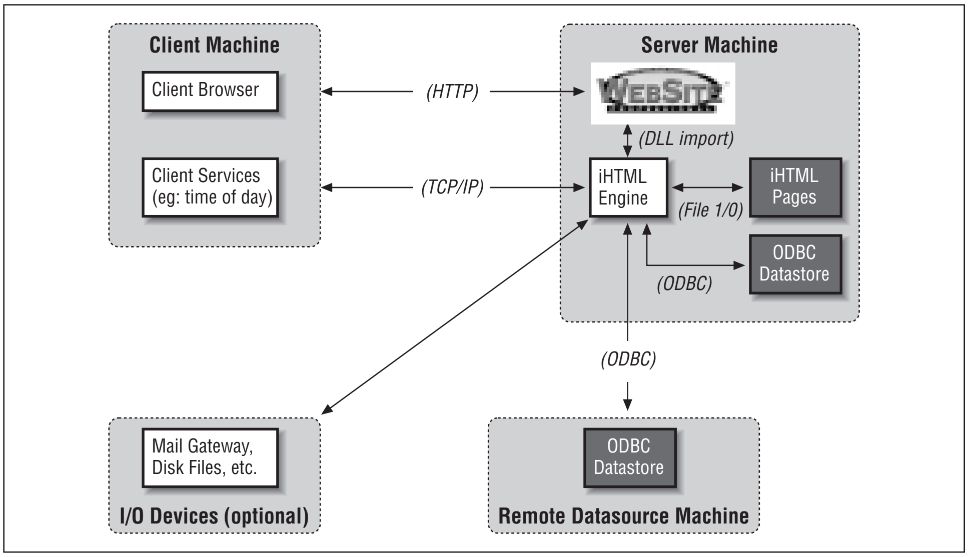
Tags and Directives

Tags are the instructions that invoke a particular command or function. They represent the commands or functions that are performed once the HTML file is requested. That is, tags are included with HTML commands to perform specific functions, such as database file queries and updates.

The iHTML tags take the following form:

```
<TAG_NAME directive1 directive 2 ...>
```

Figure 4-1 Processing an iHTML page



where each directive is a name=value pair. The order of the directives is not important in iHTML 2.1, except with the <iSET> and <iCLEAN> tags.

You must place the tag `<!ihtml>` at the beginning of an HTML page that contains iHTML tags or environment variables. Without this special tag on the page, iHTML will not get parsed. Do not use the <ihtml> tag on pages with no iHTML tags or environment variables, as it will slow the server in serving the page.

iHTML Variables

In addition to tags, iHTML resolves variables that you can include in an HTML file. There are two types of variables—environment variables and user-defined variables. Environment variables are similar to CGI variables. User-defined variables most often store form data. Together they are referred to as colon variables, because iHTML usage requires that a colon precede a variable name to identify it as a variable.

Environment Variables

iHTML 2.1 uses environment variables in place of tags for data that is normally available only from the CGI environment. You can use environment variables anywhere and when the page is processed, they return their appropriate value. Environment variables' values are derived from the following sources:

- The server
- Browser information headers

- Browser request headers
- Persistent variables (the Cookie: header field)
- Contents of GET or POST data
- iHTML engine parsing data (loops and errors)
- An SQL query

To use an environment variable in HTML, use a colon in front of the variable name. For example:

```
<!ihtml>
<BODY>
The current user is: <b>:i_auth_user </b>
The IP address is: <b>:i_ip </b>
The user's current password is: <b>:i_auth_pass </b>
The user was authenticated with <b> :i_auth_type</b> authentication
</BODY>
```

Depending on the exact conditions of the HTTP request (the browser and server used, their configuration, and the request details such as method, URL form, and so on), all variables may not always be available. Most variables are configured with either an empty string or a 0, depending on their purpose, if there is no data for them. Other variables such as :i_loop and the error-related variables, as well as variables derived from GET or POST data may be present, depending on the details of the request and the program flow. You can test to see if a variable is present with the <iISDEF> tag. The environment variables are listed and described in Chapter 6.

User-defined variables and form data

When a form is submitted in an HTML document (using either the POST or GET method), iHTML 2.1 saves the form's fields in iHTML user-defined variables. You can access these variables using the following notation:

```
:<field_name>
```

For a user-defined variable, names must start with a letter and can be followed by letters, numbers, or underscore (_) characters. Here's HTML for a form with properly named fields:

```
<FORM ACTION='somefile.html' METHOD=POST>
Name: <INPUT NAME="name" TYPE="textbox">
Age: <INPUT NAME="age" TYPE="textbox">
<INPUT TYPE="submit">
</FORM>
```

When the server receives this request, it loads *somefile.html.* iHTML passes the two fields from the sample form, *name* and *age*, to this file. The maximum length for each value in the input is 128 bytes. The values can now be accessed in *somefile.htm.* The following examples demonstrate this feature.

To display the values, type the following in *somefile.html*:

```
You entered :name for NAME and :age for AGE
```

To save specific values in a table, you could type the following in *somefile.html*:

```
<iHTML SQL="INSERT INTO table (name,age) VALUES (':name', :age)"
NOOUTPUT="Values saved"
FAILURE="An error occurred">
```

In this example, the name field is treated as a text string (because it is enclosed in single quotes in the value-list, which is the SQL method for passing string parameters), and the age field is treated as a number.

Note

iHTML 2.1 does not distinguish between numeric and alphanumeric input types. It simply parses the field references and replaces them with the submitted values. This is done *before* the query is sent to the database.

Nonexistent variables

When a variable is used and the name following a colon is not recognized by iHTML, iHTML returns the colon and the name. To check whether a user variable is defined, use the <iISDEF> tag; this tag returns FALSE for any variable that starts with i_, regardless of whether the variable is defined.

In the following example, since Fred is not a variable, it stays as :Fred in the output.

```
Name:Fred Age:<b><imath A= :age B = 1 o=+></b>
```

Assuming age is a valid variable and it is equal to 20, the above would resolve to:

```
Name:Fred Age:21
```

Multiple fields with the same name

An HTML form can contain multiple fields with the same name. In the case of a radio button, this is not an issue, since only one value is returned. However, in the case of a group of checkboxes, or a multiselect list box, it becomes an issue, since all selected values are important. iHTML handles this by separating all selected values with commas in the field’s value. This enables you to use the SQL 'IN' clause.

For example:

```
<FORM ACTION="somefile.htm">
<INPUT TYPE="checkbox" NAME="city" VALUE=1>New York
<INPUT TYPE="checkbox" NAME="city" VALUE=2>Boston
<INPUT TYPE="checkbox" NAME="city" VALUE=3>London
```

```
<INPUT TYPE="checkbox" NAME="city" VALUE=4>Dallas
<INPUT TYPE="checkbox" NAME="city" VALUE=5>Paris
....
</FORM>
```

If you select more than one city in the above example, the city field will have multiple values. Suppose you selected New York, London, and Dallas (values 1, 3, and 4). Then the :city variable would contain 1,3,4. Now if you want information on these cities, you could use the following SQL statement:

```
<iHTML SQL="SELECT info FROM table WHERE city IN (:city)" OUTPUT=":1<BR>">
```

The `:1` is the first column of the result set if it exists. The <iSQL> tag can refer to the colon variables in the result set in the same way or you can refer to the actual result columns' names. If you want to save the values to a table, you can use the following method:

```
<iHTML SQL="INSERT INTO selected_city (id)
SELECT id FROM city WHERE id IN (:city)">
```

These examples assume that an INSERT statement can take a SELECT statement as its value list.

Special character variables

iHTML has some special variables to use to output a nonprintable character. These are most often used with the <iFILE> tag.

`:$tab`	Use to output a Tab character
`:$crlf`	Use to output a carriage return and line feed
`:$space`	Use to output a space
`:$ff`	Use to output a form feed
`:$esc`	Use to output an Escape character

iHTML Markers

iHTML markers identify sections of an HTML page, for example, error handling blocks. Markers include such tags as <iERROR>, </iERROR>, <iWHILE>, </iWHILE>, <iPOP>, </iPOP>, <iCASE>, </iCASE>, <iSQL>, </iSQL>, <iLOOP>, and </iLOOP>. The scripting engine treats markers differently from other tags. Their position within the source file determines what is parsed and how it is parsed.

Carriage Return/Line Feed (CR/LF)

The OUTPUT and directives format will *not* automatically add carriage return/line feeds (CR/LF) to its output. If you require a CR/LF, include a <BR> tag in the output string. Or just extend the quotes over multiple lines in the file if you want the actual line break characters.

Note

Anything between double quotation marks in the OUTPUT or tag directives is taken literally. This is an important feature of iHTML that can be used very effectively with the <iFILE> tag. For example, if in using the iFILE command you want to write out a tab character, you simply put a tab character between double quotes in the DATA directive. The same applies for carriage returns and line feeds.

Permitted Constructions for Tags and Variables

An iHTML file consists of a sequence of tags, colon variables, markers, and text. This section gives you the rules for what you can put in a tag, permitted variable names, and other restrictions on constructing the elements of iHTML files.

Tags may contain the following:

- An HTML or iHTML tag name
- Tags
- Colon variables
- *Name* = *value* pairs
- Text

A colon variable may contain the following:

- A colon as a starting delimiter
- Whitespace or tag delimiters as an ending delimiter
- Any characters except a double quotation mark (“), whitespace, or a left or right angle bracket (< >) between the delimiters. Double quotation marks can be included if escaped by %.
- Period (.), underscore (_), and dash (-) are also valid characters for a colon variable

e = *value* pairs contain the following:

- A whitespace beginning delimiter.
- A name consisting of any characters except a double quotation mark (“), whitespace, or a left or right angle bracket (< >). Quotation marks can be included if escaped by %.
- An equals sign (=) combined with an optional whitespace delimiter.
- A value consisting of a tag; a colon variable; text consisting of any characters except a double quotation mark (“), whitespace, or a left or right angle bracket (< >) (quotation marks can be included if escaped by %); or quoted text. Quoted text contains quotation delimiters on each end, any characters

except a double quotation mark (“) or a left or right angle bracket (< >) between delimiters, tags and variables, escape characters.

- A whitespace or end of tag delimiter.

The iHTML Parsing Process

As mentioned earlier, the iHTML engine parses HTML files that contain the <!HTML> tag. The iHTML engine uses the following procedures to resolve tags and colon variables into relevant information:

- Scan the file according to the grammar rules (described later) and resolve from the inside to the outside of nested elements.
- Execute the tags in the order in which they are encountered:
  - Remove the tag or variable from the HTML document and replace it with the results of the tag’s execution or variable’s content.
  - If an error occurs, start execution in the iERROR block, if present, or else stop.
  - If the tag occurs within a quoted directive, the engine defers resolution to the point where the directive is actually used.

If the virtual paths, 404 handler, or the error handler features are used, iHTML engine uses the following procedures, as appropriate:

- Check the virtual paths for the page, and if found, load the page specified in VIRTUALPATHS in the Registry.
- Check that the page exists; if it doesn’t, load the NOEXISTPAGE specified in the Registry.
- If an error occurs, check for an error block on the page; if there is no error block, run the NOERRORPAGE specified in the Registry.

If the iHTML engine encounters the <!iHTMLR> tag, the page is processed again (see “Recursive Processing,” later in this chapter.

Grammar rules for iHTML

It’s important to understand the rules iHTML uses when parsing your iHTML pages so that you get the results you expect from the tags and colon variables you combine with the standard HTML.

Directive names are not case sensitive, and directives can be written a few ways:

```
NAME = "text with spaces"
NAME = textwithoutspaces
NAME = <tag>
NAME = :colonvar
```

```
NAME=<TAG>
NAME=":colonvar"
NAME=':colonvar'
NAME='<TAG>'
```

If a tag is inside single quote or a variable is inside double or single quotes, it is resolved each time the directive is used. If the tag or variable is not enclosed by quotes, it is resolved once when the tag is initially parsed. This lets you choose the method that suits the situation. If the tag variable is changing each time, use it within quotes. If it is not changing, do not use quotes, for faster execution. In cases where values for the HTML <INPUT> tag come from a database, you might require single quotes so that any spaces in the data get preserved. The only combination that will not work is

```
NAME="<TAG>"
```

This will put the tag name in the variable, but will not recursively resolve it unless you are using the <iEQ> tag with the EVAL directive set to TRUE. For example,

```
<iEQ NAME=stuff VALUE="<iDATE>">
```

will make a variable called `stuff` with the value `<iDATE>`. To resolve the tag as well, you would use

```
<iEQ NAME=stuff VALUE="<iDATE>" EVAL=TRUE>
```

If you wanted to keep the quotes in the variable, then use

```
<iEQ NAME=stuff VALUE="<iDATE>" QUOTES=TRUE>
```

which would create a variable with the value "<iDATE>" where the quotes are part of the value. This becomes necessary when you are populating INPUT type boxes on a form with values that have single quotation marks in them from a database.

Recursive processing

Recursive processing—embedding tags within tags—is a powerful aspect of the language. For example, `:colonvar = :colonvar` is valid as long as the first :colonvar resolves to an expected directive name. Consider the following:

```
<!ihtml>
<iEQ name=filename value=somefile.htm>
<FORM ACTION=':filename' METHOD=POST>
Name: <INPUT NAME="name" TYPE="textbox">
Age: <INPUT NAME="age" TYPE="textbox">
<INPUT TYPE="submit">
</FORM>
```

When this is sent to the browser, it will look like:

```
<FORM ACTION='somefile.htm' METHOD=POST>
Name: <INPUT NAME="name" TYPE="textbox">
Age: <INPUT NAME="age" TYPE="textbox">
<INPUT TYPE="submit">
</FORM>
```

The file called as part of the form action can be a variable. Using iIF, this could be set based on another parameter, giving maximum flexibility and programmability.

Tips for Faster Processing

To keep iHTML running fast, minimize the number of objects created on the local heap. Tags such as <iPOPHEADER>, <iSQL>, and <iDTSTRUCT> create objects that are placed on the heap. The more objects on the heap, the longer it takes iHTML to find an object that is requested. Keep these tips in mind:

- Keep SQL result sets narrow, limit the use of * wherever possible.
- Do not define unneeded variables.
- Use tags like <iDTSTRUCT> only when no other tag will do.

5

Using the iHTML Merchant

WebSite Pro comes with an ecommerce solution—the iHTML Merchant. The iHTML Merchant consists of a Microsoft Access database, pages for adding, deleting, and changing data from your browser, the storefront where you display your merchandise, the shopping basket that keeps track of what customers order, and the systems for handling business details such as shipping and taxes. Everything is ready to go. You simply modify as needed to suit your products and sales requirements.

The simple steps to installing iHTML Merchant, filling it with your products, and putting out your ecommerce Open sign, are provided in "Getting Started with iHTML Merchant." This section contains the quick start instructions for users who want to get up and running and don't need or want to understand how it all works.

The following sections expose the working of this ready-to-use component. Understanding how it all works—including administering the site and setting up the shopping basket and product catalog—will not only allow you to customize the site easily, but will also give you a jumpstart on building other iHTML solutions.

Getting Started with iHTML Merchant

Setting up the iHTML Merchant consists of installing a storefront and then using your browser to customize it for your needs. This section tells you how to install a sample version and an empty storefront. It then gives the basic instructions for customizing the site and creating your online catalog.

Installing the iHTML Merchant

To install the Merchant, insert your WebSite Pro CD in the drive and click iHTML Merchant on the opening screen. This will create a program group called iHTML Merchant which contains two icons—Storefront Setup and a shortcut to WebSite Pro Resources.

Storefront Setup gives you two choices for what to install—the sample storefront or a new empty storefront.

Merchant Sample

If you haven't already explored the sample, be sure to install it first so you have a chance to see how it works, taking a look at the administration pages and exploring the storefront. Once you've installed the sample, point your browser at *http://localhost/~wsdocs/* and follow the link named iHTML Merchant Sample Storefront. The sample is setup as a mini O'Reilly book and software catalog. It is installed into *WebSite\wsdocs\*, and access control is automatically added so that the admin functions are available only to members of the administrators group (unlike the empty store.)

Empty Store

After you've looked over the sample store, choose to install the empty store. This installation provides you with everything you need for customizing your virtual store and putting in your merchandise.

After selecting to install the empty store, you are asked for two additional pieces of information: location (where you want to install the empty store) and datasource name (the name of the database and datasource).

Choosing the location

Storefront setup suggests a default directory under WebSite's document root, for example: *c:\website\htdocs\store1\*. If you wish to install the empty store elsewhere, make sure that you select a directory location that is visible from the Web. (You may want to read up on mapping in Chapter 9 in *Mastering the Elements*). Preferably, the location you specify should be a new subdirectory below an existing web-visible directory. If the location you specify does not yet exist, Storefront Setup will create it for you.

Choosing the datasource name

This is the name you choose to identify this store's database and datasource. It must be at least 3 characters long and no longer than 24 characters. The empty store install program will append an incremented number to the end of the name you have specified, to avoid the possibility of conflicting with a name already in use as a datasource; for example: if you specified *cars*, the database and datasource name is stored as *cars1*, and so on.

Storefront Setup places all the iHTML pages for the storefront, including the administration pages, in the directory you specified. It creates a database containing all the properly named tables, set up to work with the iHTML pages and ready for you to add your products and configure your site. If you decide to install additional stores, just run Storefront Setup again, specifying a different location and datasource name—it's that simple!

Note

As noted above, the Merchant sample store sets access control so that only members of the administrators group have access to the administration screens. That isn't the case with a new empty store. To ensure proper security levels, please apply access control to the *\admin* subdirectory of your newly installed empty store.

Customizing the Storefront

Now that everything is installed, point your browser at the *index.ihtml* file in the directory you chose for the installation. You'll see your storefront's entry page, with placeholders for text and graphics. To replace the placeholders with your own information and images, point your browser at the *admin/* directory in the directory where you installed the Merchant; for example, *http://your.site.com/merchant_directory/admin.*

At the main Administration page, click Site Configuration. Use the Site Configuration page to enter site-wide specifications, such as background color for your pages, site name, and site email address. The Site Configuration page gives you instructions on the various options. At this time, you may want to work with Property Configuration and possibly Custom Properties only. If you want to understand more about how site configuration works, see "Configuring the System," later in this chapter.

Adding Your Merchandise

The iHTML Merchant catalog is based on categories and subcategories. Before adding your products to the catalog, plan out how to group them so that your customers will be able to narrow their search to find the specific item they want. For example, a housewares catalog might first categorize by bedroom, bath, kitchen, and living areas. Bedroom might be subcategorized by storage systems, linens, and floor coverings. After you have decided how to divide your merchandise into categories and subcategories, you are ready to enter the information in the Merchant. Go to the main administration page (for example, *http:// your.site.com/merchant_directory/admin)* and then click Product Management. From here you can add categories and items.

Setting Up Shipping and Taxes

You set up shipping and taxes for your site on the Site Configuration page. From the site entry page, click View the administration pages and then click Site Configuration. Choose Property Configuration and enter the information in the form provided. For information on how the Merchant calculates these costs see "Calculating Shipping Costs" and "Figuring Taxes," later in this chapter.

Deleting the Storefront

To remove a storefront, simply delete the directory in which it was installed. You'll also want to remove the storefront's datasource. To do this, go to the Control Panel and select ODBC, System DSN, and select the appropriate datasource. Once highlighted, click Remove.

How the iHTML Merchant Works

Although it is possible to use the Merchant without getting into the script that powers it, most developers will want to look under the hood, either for their own satisfaction in knowing what's going on, or because they want to make changes to the system. The iHTML code in the following sections illustrates how the administration system, the storefront, and the sales management pieces all work together to enable you to sell merchandise online. In some cases, the scripts you find in your iHTML Merchant directory differ from the examples in print. The examples are intended to explain the design and function of the Merchant pages.

7

Server-Side Scripting with Active Server Pages

Active Server Pages (ASP) is Microsoft's support framework for server-side scripting. It lets you mix HTML with in-line scripting and is fully compatible with Microsoft's ActiveX controls and components. ASP is advertised as the final puzzle piece that unifies Microsoft technologies by integrating ActiveX Data Objects (ADOs). ALthough you can use ASP to produce simple HTML pages, it is designed to tie your web pages into the data stored in databases on your own and other systems. For example, among existing freeware components designed for ASP are a credit card verifier and a conversion utility to convert Chinese characters. Others have designed ASP pages to upload files, retrieve mail, and send SMTP mail directly from a web page.

ASP itself is not a scripting language—it only provides the environment that processes your script. Microsoft developed ASP to work with its Personal Web Server (PWS) and Internet Information Server (IIS) and states their intention of incorporating it, along with their servers, into the operating system.

WebSite Pro fully integrates ASP into the server, allowing you to process and develop active server pages within the ASP framework. In fact, WebSite Pro makes it easier to handle some common tasks you encounter when scripting for ASP by providing a graphical user interface for such things as switching primary scripting languages and setting up virtual directories.

This chapter describes how to activate WebSite Pro's built-in support for ASP and gives a brief description of how the server handles active server pages. You'll find an overview of ASP scripting—just enough to get you oriented before plunging into the official ASP online documents from Microsoft. Next comes a detailed description of the server properties that affect ASP and some detailed information about settings and associations you need to have configured correctly to run ASP with WebSite Pro. You'll also find suggestions on where to go for more information on ASP.

Activating WebSite Pro's ASP Support

Before you activate WebSite Pro's ASP support, you should have ASP installed and running on your system. As a prerequisite to installing ASP, you need to have Internet Information Services (Peer Web Services, if you're using NT workstation) or Personal Web Server installed on your system. This is a Microsoft licensing requirement; in addition, you may encounter difficulties during installation of WebSite Pro's ASP support if you haven't installed one of the Microsoft servers. You should find the servers on your operating system installation CD, or you can download them free of charge from Microsoft's web site.

Note

You don't need to use the Microsoft server once you've installed ASP and activated WebSite Pro's ASP support.

After installing the Microsoft server, you can install Active Server Pages, which is a free, integrated feature of Windows NT and Windows 95. (Look in the IIS download area on Microsoft's web site for ASP.) After it's installed, you may want to explore the sample ASP applications that come with the installation and look through the tutorial.

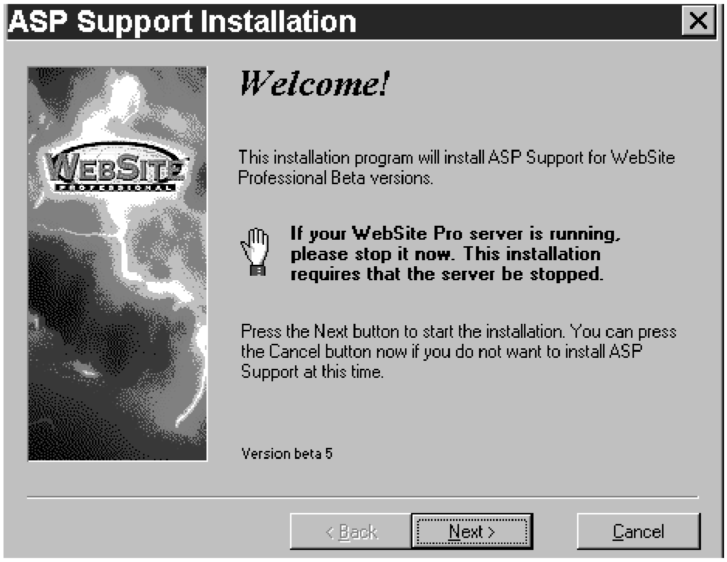
ASP has native support for JScript and VBScript. Although the ASP documentation and examples provided by Microsoft are strongly slanted toward the use of VBScript, the architecture allows you to script with other languages. If you choose to develop scripts or run scripts in languages other than JScript and VBScript, you need to install their scripting engines. As of this writing, third party plug-ins provide support for Perl, REXX, and Python. Perl and Python come with WebSite Pro. You'll find their installers on the WebSite Pro CD.

Since ASP documents are essentially HTML with additional tags containing script language, you can use a simple text editor, such as Notepad, or an HTML editor, such as HomeSite, to create them.

Running the ASP Support Setup Wizard

Once you have ASP installed and you've tested it enough to be sure it's working properly with IIS or PWS, you're ready to activate the built-in WebSite Pro support for ASP. The WebSite Pro ASP Support Setup wizard handles the necessary configuration (Figure 7-1).

Figure 7-1 WebSite Professional ASP Support Setup Wizard



ASP Support Installation

Welcome!

This installation program will install ASP Support for WebSite Professional Beta versions.

If your WebSite Pro server is running, please stop it now. This installation requires that the server be stopped.

Press the Next button to start the installation. You can press the Cancel button now if you do not want to install ASP Support at this time.

Version beta 5

Shut down all running applications and insert the WebSite Pro CD. Click Active Server Pages (ASP) Support to start the ASP Support Installation Wizard. After setup is complete, you are instructed to start your server to view the ASP Roadmap. If you prefer, you can view it later by double-clicking the ASP Roadmap icon on your desktop. If you encounter any difficulties, check “Getting Essential Settings Right” later in this chapter.

Note

You may be asked for a username and password before you can view the ASP Roadmap. Use the WebSite Pro administrator account and password. All of WebSite Pro’s samples and test pages require authentication. This is to prevent security holes that might otherwise exist.

15

Programming WebSite Pro with Java

Java is a hot topic in computing today—especially when talk turns to the Internet or intranets. Java is a programming language that supports networked applications and enables developers to write applications that will run on any Java-enabled machine. Developers praise Java as a powerful and flexible software platform that promises to transform web development, breaking down barriers between different computer hardware and software systems. Java servlets—server-side Java applications—allow you to write Java programs to extend and amplify your server's features.

Java was conceived by a development team at Sun Microsystems looking for a small, reliable, architecture-independent language to program consumer electronic devices such as microwave ovens and personal digital assistants (PDAs). When the World Wide Web appeared on the scene, the team immediately saw that Java was ideally suited to programming on the Internet. Here's why:

- Java is a simple language with a limited number of language constructs. It is similar to C and C++, but doesn't use or need pointers.
- Java is object-oriented, which lets developers focus on data and data manipulation.
- Java is a distributed language, which means it supports various levels of network connectivity through its classes.

- Java is an architecturally neutral language because it is an interpreted language. The Java compiler generates byte code—not machine code. Byte code provides an architecturally neutral object file format that can be transported easily to various platforms and interpreted by the resident Java interpreter.
- Java was designed for writing reliable software. The language avoids some of the most common danger areas in writing code.
- Java is a secure language, which is especially important in networked environments.
- Java provides high performance. Even though as an interpreted language, it is not as fast as a compiled language, it is more than adequate for GUI and network-based applications. Java is faster than scripting languages, such as VBScript or Perl, overall.
- Java is multithreaded and dynamic.

Today most web browsers support Java applets, small Java programs that are embedded in the HTML documents. The Java virtual machine integrated with the browser executes the applet and controls its area in the browser display. A common, trivial, use of Java applets is to create graphic animations. But Java is capable of much more. For example, a Java applet could obtain stock quotes from a remote site and display them in the browser. Or the applet could provide local forms with live field validation. Most developers are familiar with applets, and Java is strongly associated with browser-side applications.

You can also use Java to create complete applications. Increasingly, companies are employing servlets to extend the capabilities of their web sites. For example, InfoSpace in San Mateo, California, integrated their databases into the Internet and intranets using a data access program based on Java. OpenConnect Systems in Austin, Texas, created Java-based tools to address a common challenge to business—integrating legacy hosts, such as mainframe, midrange, and minicomputer systems into the infrastructure.

The WebSite Pro Java Servlet Software Development Kit (SDK) facilitates servlet development, letting you tap into Java to add innovative and practical features to your server. The remainder of this chapter introduces the WebSite Pro Java Servlet SDK, explaining how it differs from the previous Java CGI programming package. You'll get the tools and concepts you need to use the SDK efficiently. Examine and test the sample servlet code, and you'll be on your way to developing your own servlets. A final section on troubleshooting should help if you run into any snags.

Preparing to Use the SDK

The WebSite Pro Server-Side Java SDK provides a framework for developing Java servlets—Java server-side applications that run in the WebSite Pro program space.

These servlets use the WebSite Pro internal Java interpreter, which means you're assured of fully overlapped, multithreaded, very fast execution. The SDK includes:

- The foundation classes for servlet Java programming with WebSite Pro. The foundation classes are in the Java class package, *WebSite.Servlet*. These classes support all WebSite Pro's WSAPI features. This means your servlet can automatically decode forms, both URL-encoded and RFC 1867 forms. There's a class to provide state-management "session" capability, eliminating the need for you to manually handle cookies and expiration. For those who choose to manually handle cookies, there's full support for Netscape and RFC 2109 cookie formats. Another class provides access to the browser capabilities database maintained in the *browscap.ini* file. In addition, a global exception handler assures that unhandled Java program errors are caught and reported to the browser.
- A Windows dynamic link library (DLL) that integrates the Java interpreter into the WebSite Pro process space and provides the interface between Java and WSAPI. This is described in detail in "How WebSite Pro Runs Java Programs," later in the chapter.
- The Java Runtime Environment to enable you to run Java servlets without installing the Java Development Kit.
- Complete API-style documentation for the SDK foundation classes. The WebSite Pro Java Servlet API is organized and documented on-line using the same style and conventions as the JDK 1.1 API. The foundation classes are also fully documented in the *Java Servlet SDK Classes Reference*.

Note

For generic information on how to use the on-line documentation, see the *JDK 1.1 API User's Guide* that comes with the Java Software Development Kit 1.1.

- Example Java programs to help you get started quickly. Run *WSJava.servlet* (*http://localhost/wsjava.servlet*) and then look at the source code. This servlet demonstrates most of the features provided by the WebSite Pro Java Servlet SDK. You can also check out the source code for the *Pizza.servlet*.

Installing the WebSite Pro Java Servlet SDK

The Java SDK Setup Wizard handles the details of installing the WebSite Pro Java SDK automatically. Simply insert the WebSite Pro Installation CD and click Java SDK to start the Java SDK Setup Wizard. If you have not installed the JDK 1.1, the Wizard installs the Java Runtime Environment (JRE 1.1). If you have installed the JDK 1.1, the Wizard checks that the JDK path information it has is correct. After the Wizard completes the installation, start the server and you'll be taken automatically to an online test of the WebSite Pro Java Servlet SDK.

If you install the Java Runtime Environment and later install the JDK, you can simply enter the path to the JDK root on the Java tab of the server's property sheet. See "Switching from JRE to JDK" later in this chapter.

Changes from WebSite Pro Java CGI Programming Package

The WebSite Pro Java Servlet SDK supersedes the server-side Java CGI programming package that shipped with WebSite Professional in May 1996. The Java runtime engine has come a long way since then, and the new Java Native Interface (JNI) supports integration of the Java interpreter into other processes. So, unlike with the original CGI-based package, the Java interpreter is now integrated into WebSite Pro, and provides access to many of WebSite Pro's WSAPI features.

Even though the underlying technology is very different, the programming interface has changed very little from the original *WebSite.WinCGI* package. In most cases, all you need to do is change your import statements to import from the *WebSite.Servlet* package, and change `extends CGI` to `extends Servlet` in your main class. The class proxy files for this package have the extension *.servlet* instead of *.clx*.

Setting Up Your Development Environment

In order to write servlets for WebSite Pro, you'll need to install and become familiar with the Java language and development environment. Don't be alarmed; writing servlets is easier than applet programming. The easiest development environment to obtain is the Java Developer's Kit (JDK) 1.1 for Win32, which is available free for personal use from Sun. Another excellent Java development environment is Symantec Visual Café. There are links to these sites at WebSite Central.

If you choose to use Microsoft's Visual J++, be warned that some restrictions apply. Be sure not to use Microsoft proprietary OLE, COM, or ActiveX features in your Java servlet. These features are supported *only* by the Microsoft Java Virtual Machine, which is not JNI compliant. WebSite Pro requires a JNI-compliant interpreter, so you can't use the Microsoft Java interpreter, and WebSite Pro's JNI-compliant interpreter won't run your Java servlet if it includes any OLE, COM, or ActiveX extensions. Of course, you can use Java Beans components in servlets.

There are several additional steps you can take to make your Java programming experience more pleasurable and productive. Your time investment setting up will be amply rewarded when you start developing servlets.

- Get a copy of O'Reilly & Associate's book *Java in a Nutshell* (*http://www.oreilly.com/catalog/javanut*). One reviewer wrote, "This book is an indispensable quick reference designed to lie flat and wait faithfully by the side of every Java programmer's keyboard. It contains an accelerated introduction to Java for C and C++ programmers who want to learn the language *fast.*" The new May 1997 edition contains an update for JDK 1.1. O'Reilly also publishes other books documenting Java. Check out the publications at *http://www.oreilly.com/publishing/java/products.htm*.

- If you aren't using Symantec Café or Microsoft Visual J++, find a text editor that understands the C/C++ language structure. Java is so close to C and C++ in its syntax that C-aware editors work very well with Java. There are a number of programmer's editors that are language-sensitive to C, including Programmer's File Editor and the Win32 port of GNU EMACS. If you do choose to use GNU EMACS, you can get an EMACS package that supports Java editing specifically. You'll find links to these tools at WebSite Central.
- If you aren't using an integrated development environment such as Symantec Café or Microsoft Visual J++, get the 4DOS/NT command shell, which works wonderfully on Windows 95 as well. A real shell is extremely useful when working with Java. The cost is minimal ($69 as of our publication date) and you can download the software for trial. Don't download from a BBS or other unofficial site. Your shell is your life. Get it directly from JP Software, using the link at WebSite Central. Be sure to download the 2.51 version and the Rev-B patch (or later version if available).

  On Windows NT, set up 4DOS/NT to run in a console window of 40 lines height and at least 200 scrollback lines. This makes it possible to see all your compilation errors (in case you have any!). On Windows 95, you can't set scrollback on console windows. You can use the 4DOS `list` command to page the output of the compiler (pipe the compiler output to `list`) and create a command alias that does this automatically. The `list` command lets you page up and down in the compiler output, which is very useful. Set up other aliases for things like your editor and the applet viewer.

After you get everything set up, start out by compiling a few of the JDK sample applets, and play with them using the applet viewer or a browser that supports Java. This will give you a feel for the development environment and you'll probably have some fun too!

How Java Finds Classes

Java combines an environment variable with strict naming conventions to give the class loader all the information it needs to find classes. Before looking at the naming conventions, be warned that one aspect of Java that can trip you up if you're not paying attention is that Java is case sensitive. If you create a class named Menu.cats and tell Java to look for the Menu.Cats class, it won't find it. Keeping Java's case-sensitivity in mind, let's examine the important CLASSPATH environment variable.

The CLASSPATH environment variable tells the Java class loader in which directories to look for classes. For example, if you installed the JDK in *C:\JDK*, CLASSPATH would be defined as:

```
.;C:\JDK\Lib;C:\JDK\Lib\Classes.zip
```

Notice that the current directory is included in the class path. (This is so that during development the class loader looks first in the directory you're developing in and loads your *.class* files.) The *lib* directory contains classes not in the *Classes.zip* file, which contains the core Java classes. (Do not unzip this file!) Depending on what you're doing with Java, you might add directories containing classes you've defined to CLASSPATH.

WebSite Pro's Private Class Path

Most, but not all, Java programs use the CLASSPATH environment variable to locate Java classes. The "not all" disclaimer is especially important when talking about the WebSite Pro Java Servlet SDK, because WebSite Pro's embedded Java interpreter uses a private class path to look for classes. You can see it on the Java tab of the server's property sheet. For example, if WebSite Pro is installed in *C:\WebSite*, and the JDK is as above, the value for CLASSPATH would be:

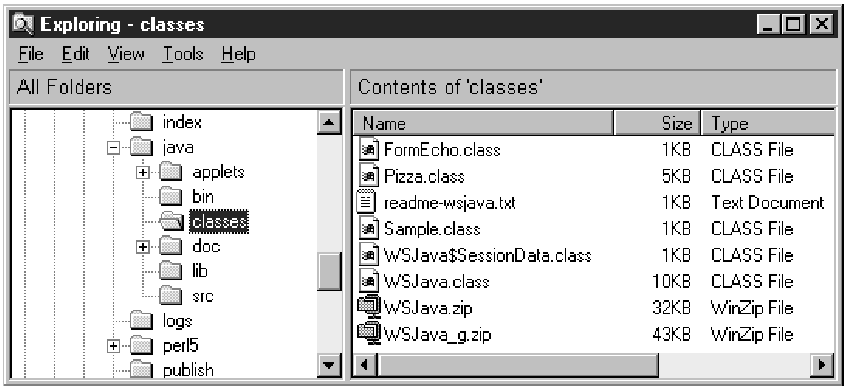
```
C:\WebSite\java\classes;C:\JDK\Lib;C:\JDK\Lib\Classes.zip
```

WebSite Pro uses the private class path so that you can freely alter the CLASSPATH environment variable during Java application development without having to worry about causing problems with the server. Of course, it also makes it possible to develop WebSite Pro servlets without disrupting any of your other Java work.

Let's look at the first path stored in WebSite Pro's private class path. The *classes* directory is where you put your servlets and any Java packages. A package is a group of related classes and is an essential part of the Java class naming conventions, which enable Java to use class names to locate the class files. More on this soon.

For now, take a look at what was installed in the *classes* directory that serves as the first root for the class path (Figure 15-1). It contains various files. The files with a *.class* extension are Java executables (you'll meet the FormEcho class later in the examples). There are also two ZIP files. *WSJava.zip* contains the regular version of the *Website.Servlet* package with the Java Servlet SDK foundation classes. *WSJava_g.zip* contains the debugging version. Java developers often use ZIP or JAR files to manage Java packages with many class files.

Figure 15-1 A class path root directory



Name	Size	Type
FormEcho.class	1KB	CLASS File
Pizza.class	5KB	CLASS File
readme-wsjava.txt	1KB	Text Document
Sample.class	1KB	CLASS File
WSJava$SessionData.class	1KB	CLASS File
WSJava.class	10KB	CLASS File
WSJava.zip	32KB	WinZip File
WSJava_g.zip	43KB	WinZip File

Class Names and Packages

The class name actually contains information about the class file location. In the simplest case, Java class names are relative to one of the root directories in the class path and describe the path to the class file. For example, *Pizza.class* is in the *classes* directory. However, Java also uses packages as a way of organizing and manipulating class files. A class that is part of a package can be referred to by a name that's relative to the package itself, not to a root directory in the class path.

Let's look at the *WebSite.Servlet* package. The dotted package name indicates that *Servlet* is a subdirectory of the *WebSite* directory, which is in one of the root directories of CLASSPATH—in this case, *C:\WebSite\java\classes*. Of course, you don't see these directories in the Explorer, because they have been zipped up into the *WSJava.zip* file. If you open the ZIP file with a utility that allows you to see the contents, such as WinZip, you'll see all the classes in the *Website.Servlet* package. Don't unzip it, just look at its contents! On the right, in the Path column, you'll see the WebSite and Servlet directories.

On the left are the classes in the *WebSite.Servlet* package. You can see that the class names don't describe the full path from the CLASSPATH root. To get the full class name, you combine the class name with its package name. One of the classes in that package is named *Form.class*, so its full name is WebSite.Servlet.Form.class.

When the Java interpreter comes to a class to be loaded, it converts the class name into a pathname, in this case, *WebSite\Servlet\Form.class*. The pathname is a relative pathname and is relative to the directories in the CLASSPATH variable. Java steps through the directories until it finds the class and then uses the full pathname to load the class. In the example, the full path is *C:WebSite\java\classes\WebSite\Servlet\Form.class*.

The *classes* directory is intended as the starting point for you to use in organizing the servlets (and any packages) you create for the WebSite Pro server. If you do choose to place any of your Java classes in other locations, make sure to adjust the WebSite Pro private CLASSPATH value in the Registry to include the root directory of your chosen location. Otherwise, the Java class loader won't be able to find your class files.

Creating Java Class Archives

As you've just seen, Java class packages are commonly kept in ZIP files. The JDK 1.1 keeps its core classes in a ZIP file — and provides plenty of warnings about not trying to decompress the file. The ZIP archive actually contains a directory that maps to the class files contained in the archive. This enables you to group large numbers of classes. You can also store your own classes in ZIP files. Do not use compression in creating your ZIP file.

You can also keep everything you need for your servlet in a JAR file. A JAR file is a Java archive that combines all of an Java application's class files, sound, and images.

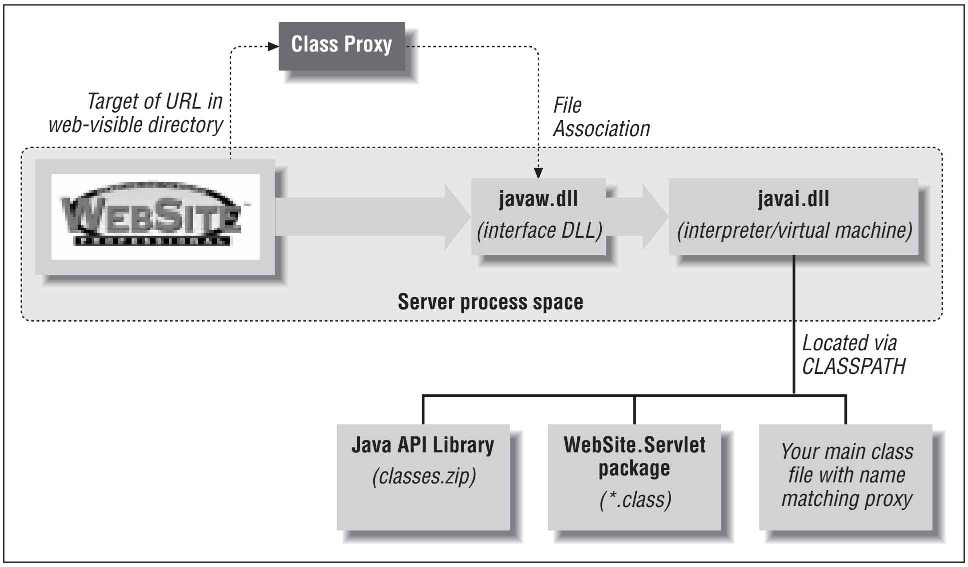
How WebSite Pro Runs Java Programs

The WebSite Pro Java Servlet SDK consists of a Java class library, the WebSite.Servlet package, and the WebSite Pro Java dynamic link library (*wsjava.dll*) that is written to work with WebSite Pro's WSAPI interface. When the WebSite Pro server is first started, it loads the *wsjava.dll* which in turn loads the interpreter *javai.dll*. The way that WebSite Pro, WSAPI, and Java work together is a bit unusual. The diagram in Figure 15-2 should help you to understand how these three relate to each other.

During the WebSite Pro installation, the installer associates *.servlet* with the WebSite Pro Java DLL *wsjava.dll* so that the operating system knows to open any files with the *.servlet* extension with *wsjava.dll*. The installer also adds a server-side content-type mapping for *servlet* that maps it to a WSAPI extension (*wwwserver/wsapi*). This tells the server to use WSAPI when it processes the *.servlet* file. Keep these two facts in mind.

To run a Java servlet program, you need to put a class proxy file — a specially named file with the *.servlet* extension — into a web-visible directory. The default web-visible directory for WebSite Pro is *C:\WebSite\htdocs* (the document root). The name of the class proxy file should be *classname.servlet*. For example, you would put the *Pizza.servlet* class proxy in *C:\WebSite\htdocs*. The extension (*.servlet*) is important only for the association to the *wsjava.dll* extension. The contents of the proxy file can be anything, it just cannot be empty. Fill it in with “servlets are cool” or something like that.

Figure 15-2 Java and WebSite Pro integration


Class Proxy
Target of URL in web-visible directory
File Association
WEBSITE
javaw.dll (interface DLL)
javai.dll (interpreter/virtual machine)
Server process space
Located via CLASSPATH
Java API Library (classes.zip)
WebSite.Servlet package (*.class)
Your main class file with name matching proxy

When WebSite Pro receives a URL directing it to the class proxy, it invokes *wsjava.dll*, which has been associated with the *.servlet* extension. The DLL strips away the *.servlet* extension and uses the resulting class name as the primary class to load for the servlet. Now, remember the content-type mapping that maps the *.servlet* extension to WSAPI? This tells the DLL to send the class name to the Java class loader through the WebSite Pro API. As described earlier, the class loader uses the name of the class proxy file to find the class for Java to start, using the WebSite Pro private class path to search for the corresponding *.class* file. It then creates a new object of that class. This is your new servlet. The WSAPI extension then performs an intricate series of calls and callbacks to attach the WebSite Pro worker thread to the Java virtual machine and establish the proper contexts for the *Website.servlet* package static classes (Client, Request, and so on). Finally, *WSJAVA.DLL* calls your `run()` method and your servlet is running.