APPENDIX B

Extending Dreamweaver

IN THIS APPENDIX

The beauty in Dreamweaver doesn't lie solely in the fact that you can build large-scale websites and applications; it also lies in its interface. *Interface*, you ask? Dreamweaver, unlike many applications on the market today that force you to purchase costly third-party extensions and work in a rigid menu-driven environment, allows for complete customization and control by the user. A relatively new concept in software development, Dreamweaver was developed with user workflow and task achievement in mind, regardless of how you prefer to work.

Because Dreamweaver's menus, dialogs, objects, and commands are built on a JavaScript/HTML/XML foundation, all are completely customizable with just a little knowledge of the markup and scripting languages. This appendix focuses on extending Dreamweaver's integrated development environment (IDE) by allowing you to customize your workflow and produce the following:

- Custom objects
- Custom client-side behaviors
- Your own public extensions
- Custom server behaviors

Working with Objects

Part of the customization initiative includes objects. *Objects* are HTML files that use JavaScript to insert a string of HTML code into the user's workspace. As you might have already noticed, Dreamweaver comes installed with a host of

predeveloped objects, completely ready for you to take advantage of. Ranging from tables to frames, forms to head content, and scripts to characters, objects make adding code to your workspace as easy as clicking a button. As you'll see throughout this appendix, the process is relatively straightforward. Essentially, you construct an object using HTML, JavaScript, and a bit of XML, package it up as an installer file (MXP), and install it into Dreamweaver. Then you can use the functionality that you build by referencing the object from the Insert menu. The next few sections will walk you through the process of understanding and working with custom-built objects.

Understanding Objects

As previously mentioned, Dreamweaver comes preinstalled with ready-to-use objects. Those objects, like most of Dreamweaver's configuration files, are located within the Configuration folder of the Adobe program directory, or in most cases in `C:\Program Files\Adobe\Adobe Dreamweaver CS4\Configuration\` (`/Applications/Adobe Dreamweaver CS4/Configuration/`). From that folder you can navigate to and open the Objects folder. The Objects folder contains a list of subfolders corresponding to the specific tab within the Insert panel.

As you add folders, more tabs are added to the Insert panel. Within the sections are three files that make up the structure of an object:

- **HTML file**—The HTML file is the front end for what the object will do. It's the interface for the functionality of the object.
- **JavaScript file**—The JavaScript file contains all the client logic that the HTML page will use when the button is pressed from the Insert panel. It is optional to use the JavaScript file because the code could be contained within the `<head>` tag of the HTML file. However, a separate JavaScript file is always easier to maintain in the long run.
- **Image file**—The image file is a standard GIF image that you can customize and place in your Custom Objects folder along with the HTML and JavaScript files. The GIF image (which must be named the same as the HTML and JavaScript file) resides in the appropriate section in the Insert panel and represents a clickable link to your object. A default GIF image (`generic.gif`, located in the root of the Objects folder) is available for you to customize using your favorite image editor.

Dreamweaver objects range from fairly simple to complex. You can make it so that when users click the object (either from the Insert panel or the Insert menu), Dreamweaver inserts plain HTML text, or you can make it so that when users click the object, Dreamweaver prompts them with a dialog box allowing them to input data. Either way, objects streamline the way you work with Dreamweaver.

Objects are composed of five key elements:

- **The files**—As you saw in the previous bullet points, objects consist of an HTML page, an image file, and an optional external `.js` file that contains all the client logic. Note, however, that the three files must be named the same. If you have

`object.html`, you must have `object.js` and `object.gif` for the custom object to work correctly.

- **Location**—Objects reside in their corresponding folder in the Objects folder. Depending on which tab you want your object in, you move all the files to that folder. As I've mentioned, though, you can create your own folder within the Objects folder as well. Doing this adds a new tab to the Insert panel.
- **Page title**—The page title is the name of the object as it appears when your mouse rolls over the corresponding object's image icon within the Insert panel.
- **`objectTag()` function**—The `objectTag()` function returns the value of what is to be inserted into the page. This function is one of many functions that you can use when designing custom object functionality, also known as the extensions API.
- **User interface**—The user interface (UI) resides within the `<body>` tag of the object and generally uses a `<form>` tag along with a text box control to capture user input.

TIP

API stands for *Application Programming Interface*. APIs are the building blocks for programs, and they are what developers program against in almost all software development environments, including Dreamweaver.

The Simple <sup> Tag Object

Now that you have a basic understanding of how objects are constructed, let's put everything to work by creating a simple object that inserts the `<sup>` tag into your HTML.

If you've worked with special characters (the trademark symbol, for instance), you know that whenever they are inserted, they always end up looking like part of the text rather than superscripted. To alleviate that problem, a simple object could be created that allows the user to superscript a special character like the trademark symbol. To create this functionality as an object, follow these steps:

1. Begin by making a copy of `generic.gif` (located in the root of the Objects folder) and place it into the Text folder. Rename the GIF file **`sup.gif`**.

NOTE

The `<sup>` tag is a text-level formatting element. Because of this, it is placed into the Text folder. However, as noted previously, you could just as easily create your own folder, in which case a new tab is created along with the custom object within the Insert panel. For simplicity's sake, we'll add our custom object into the existing Text folder.

2. Create a new HTML document in Dreamweaver and write the following code:

```
<html>
<head>
<title>SUP</title>
<script language="JavaScript">
function objectTag() {
     return "<sup></sup>";
}
</script>
</head>
<body>
</body>
  </html>
```

3. Save the file into the Text folder, naming it **sup.html**.
4. Hold down the Ctrl/Option key and choose Reload Extensions from the Insert panel's category list, similar to Figure B.1.

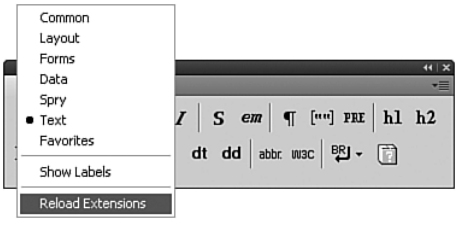

FIGURE B.1 Select the Reload Extensions option from the Insert panel's category list.

5. Create a new page in Dreamweaver by choosing File, New. Choose the HTML option from the Blank Page category, select the <none> option from the Layout list, and click Create. Immediately save the file as **sample.html**.
6. Choose the Text option from the Insert panel's category submenu. Note that the new SUP option appears within the Text category of the Insert panel, similar to Figure B.2.

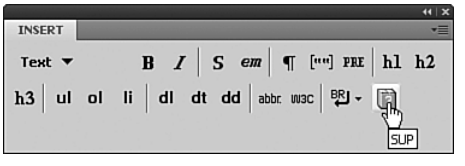

FIGURE B.2 The new SUP object appears within the Text category of the Insert panel.

7. Switch to Code view and click the new object. Figure B.3 shows how clicking the object inserts the correct tags that you specified within the `objectTag()` function outlined in step 2.

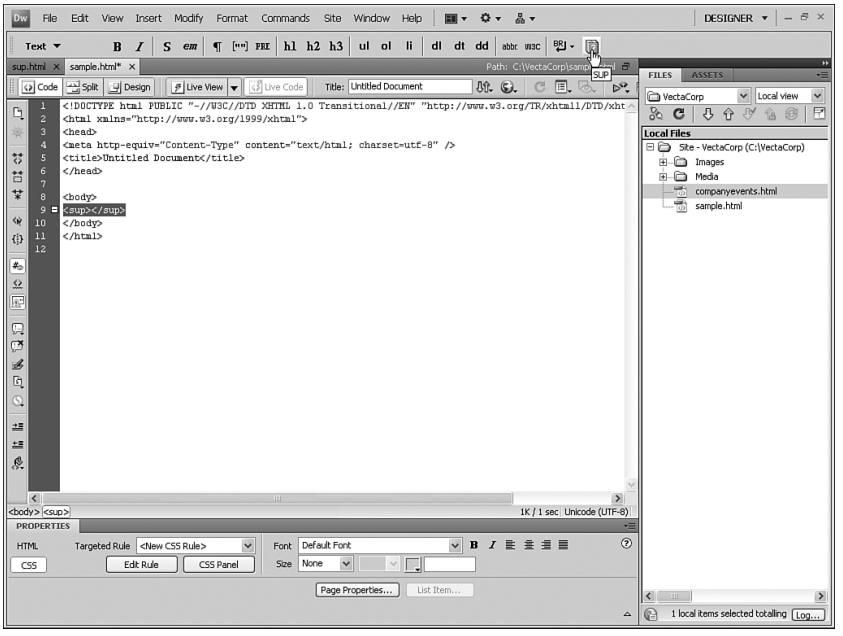

FIGURE B.3 Selecting the new object inserts the appropriate tags.

Now it's just a matter of adding text in between the `<sup>` tags to make that text superscripted.

The Advanced <sup> Tag Object

After you've gotten past the excitement of creating your first object, you'll quickly find that the object does no more than insert a simple tag into the body of the document. Even if you wanted to place some text within that tag, it would still be necessary to switch to Code view and type it in; by that time, the whole tag could have been written manually without the use of a custom object.

The power behind customized objects is that they can receive user input. By modifying the body tag to contain some HTML code (in the form of form objects) that defines the UI, your object can also receive user input. To add this functionality, follow these steps:

1. Open the `sup.html` file again if it's not already open and rewrite the `<body>` tag to contain the following code:

```
<body>
<form name="supForm">
<p>What text would you like to superscript?</p>
<p><input type="text" size="10" name="supText"></p>
</form>
  </body>
```

2. Modify the `objectTag()` function so that it concatenates the value of the text box to the opening and closing `<sup>` tags:

```
function objectTag() {
     var supText = document.supForm.supText.value;
     return '<sup>' + supText + '</sup>';
  }
```

3. Save the file, close it, and then reload the extensions in Dreamweaver again by holding the down the Ctrl key and choosing Reload Extensions from the Insert panel's category menu.
4. Open (or switch to) `sample.html`, switch to Design view, and immediately add the text **`Copyright 2007 Vecta Corporation`** within the page.
5. Place your cursor just after the Vecta Corp text and select the `SUP` object from the Insert panel. This time, rather than simply inserting the `<sup></sup>` markup in the code editor, a dialog box similar to the one shown in Figure B.4 appears.

FIGURE B.4 Selecting the new object launches a dialog box that allows you to enter custom text to superscript.

6. Type the text **`TM`** into the text box and click OK. The trademark symbol appears as superscripted in the Document window just to the right of the Vecta Corp.

Working with Behaviors

In Chapter 8, "Using Behaviors," you became familiar with some of the prewritten JavaScript capabilities that Dreamweaver offers. You were able to manipulate your application to include client-side logic that was already developed for you. But what if you need to accomplish a task that isn't on the list of preinstalled behaviors? Would you be out of luck? The answer is no!

Because of Dreamweaver's customizable behaviors list, behaviors can be created and removed as easily as they are added to your workspace. You know how to use behaviors, and might even know a little about how they work, but understanding how to create custom behaviors lies first in understanding what a behavior is and what it is composed of.

As was mentioned in Chapter 8, a *behavior* is a chunk of code that gets inserted into your working environment when you select it from the Behaviors panel list. Instead of adding plain HTML code, a behavior generally inserts JavaScript code, complete with an event and a resulting action.

As was the case with objects, creating behaviors is possible because of an extension API that outlines procedures and functions, which, in essence, allow you to insert exactly what you want into your workspace as long as you use Dreamweaver's predetermined functions (such as the `objectTag()` function that you used when working with custom Objects).

From your experience with behaviors thus far, you should surmise the following:

- Behaviors create two chunks of code: a function that performs the action you have created and a function call or event (`onClick`, `onMouseUp`, and so on) from the object you have inserted.
- You can edit behaviors by double-clicking the behavior itself in the Behaviors panel. Notice that when you do so, the dialog box for the specific behavior launches, and the original values you inserted appear in their respective editable text boxes.
- Behaviors allow you to choose the event handler that will invoke the function call.

Understanding Behaviors

Like most software applications, Dreamweaver has a standard procedure for how it handles plugins. In this case, Dreamweaver is handling much more than a simple plugin; it's handling code that you write and customize for multiple people to use. Imagine the complexity. In this case, however, it's not all that complex if you can remember some basic elements that make up the location and structure of a typical Dreamweaver behavior:

- **The file**—As was the case with objects, behaviors consist of an HTML page that makes up the user interface and an optional external `.js` file that contains the code to be processed. With behaviors, though, there's no associative image file because they're selected from a list in the Behaviors panel.
- **Location**—Behaviors reside in the Actions folder, which resides in the Dreamweaver directory. The full path usually reads `C:\Program Files\Adobe\Adobe Dreamweaver CS4\Configuration\Behaviors\Actions\ (/Applications/Adobe Dreamweaver CS4/Configuration/Behaviors/Actions/)`. Navigate to that directory and notice that the names of the files resemble the names of the behaviors in the Behaviors panel. Adding a folder in the directory creates a submenu within the behaviors list.
- **Page title**—The page title is the name of the behavior as it appears in the behaviors list.
- **Defined function**—The defined function is the code that will be inserted into your document.
- **`behaviorFunction()` function**—The `behaviorFunction()` function is part of the extensions API and is required typically just below the defined function. The

`behaviorFunction()` function returns the name of the defined function without the parentheses.

- **`applyBehavior()` function**—Also part of the extensions API, the `applyBehavior()` function returns the name of the function to be inserted into the workspace environment.
- **User interface**—The user interface is what the person selecting the behavior sees when selecting the behavior from the list. Typically, when creating custom behaviors, you want people to enter some sort of value that is, in turn, passed on to the defined function. The UI resides within the `<body>` tag of the HTML page.

The Simple Resizer Behavior

The best way to understand how a behavior works is to demonstrate how a behavior is written. The key elements that make up a behavior have been introduced to you; now it's just a matter of putting them together.

1. Begin by navigating to the Actions folder located in the Behaviors folder and create a new folder called **`Custom Behaviors`**. Remember, by creating a folder within the Actions folder, you are effectively creating a submenu to store custom-built behaviors. This folder will also appear as a submenu on the behaviors list.
2. Open Dreamweaver if it's not open already and create a new HTML file. Immediately save it into the Custom Behaviors folder and call it **`windowResizer.html`**.
3. Add the title **`Window Resizer`** to the document. This is the name as it will appear in the behaviors list. To create the functionality, you must start by adding script blocks within the `<head>` tag and add the `resizeWindow()` function within the `<script>` tag of the document (shown in bold in the following listing). The `resizeTo()` method of the `Window` object will be used along with the width and height values that it accepts as parameters to resize the browser window.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>Window Resizer</title>
<script language="JavaScript">
function resizeWindow() {
     window.resizeTo(400,400);
}
</script>
</head>
<body></body>
  </html>
```

4. Add the `behaviorFunction()` and the `applyBehavior()` functions, returning the name of the `resizeWindow()` function in both cases. Remember that the `behaviorFunction()` function does not return the `()` at the end of the function name.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>Window Resizer</title>
<script language="JavaScript">
function resizeWindow() {
window.resizeTo(400,400);
}
function behaviorFunction() {
return "resizeWindow";
}
function applyBehavior() {
return "resizeWindow()";
}
</script>
  </head>

<body></body>
  </html>
```

5. Now that the logic of the behavior has been taken care of, you are ready to begin working on the user interface. In this case, you will create a simple dialog box that displays a message to users, alerting them of the window resize:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>Window Resizer</title>
<script language="JavaScript">
function resizeWindow() {
window.resizeTo(400,400);
}
function behaviorFunction() {
return "resizeWindow";
}
function applyBehavior() {
  return "resizeWindow()";

}
</script>
  </head>

<body>
This behavior will resize<br />
the browser window.
</body>
  </html>
```

6. Save the file. If you haven't done so, open the Behaviors panel by choosing Window, Behaviors. Dreamweaver may or may not recognize that a new behavior has been added to the Behaviors folder and present you with a Refresh button. If it does present the Refresh button, click it now to refresh the Behaviors panel. If it doesn't, restart Dreamweaver. After it is restarted, the panel will have been refreshed and the behavior will appear. From my experience, I almost always have to restart Dreamweaver to get it to recognize the new category in the panel.

Congratulations—you've successfully created your first behavior! To use the behavior, we'll simply need to refresh the Behaviors panel, create a new HTML page, and add the new behavior to an element (preferably a link) on the page. You can accomplish this by following these steps:

1. Create a new HTML file by choosing File, New. Choose the HTML option from the Basic Page category, select the <none> option from the Layout list, and click Create. Immediately save your file as **`sampleBehavior.html`**.
2. Add a link to the page. You can add the pound (#) symbol to represent the link path.
3. Now highlight the new link and apply the Window Resizer behavior by choosing the Window Resizer option from the Custom Behaviors submenu in the behaviors list, also shown in Figure B.5. The Window Resizer dialog box appears. Click OK.

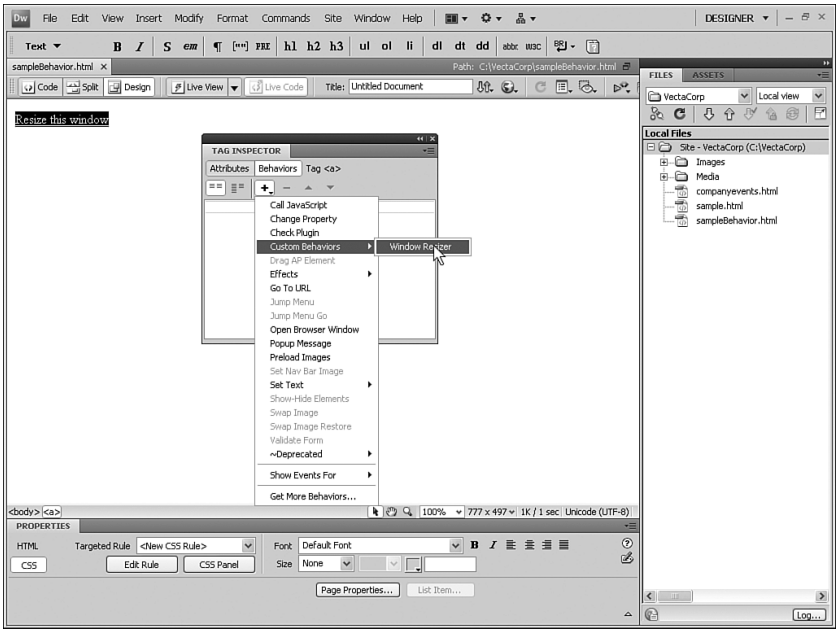

FIGURE B.5 Apply the new Window Resizer behavior to your link.

4. After you have added your new behavior, examine the code by switching to Code view. Figure B.6 shows how the function was added to the <head> tag. An event handler that calls the function was also added to your link.

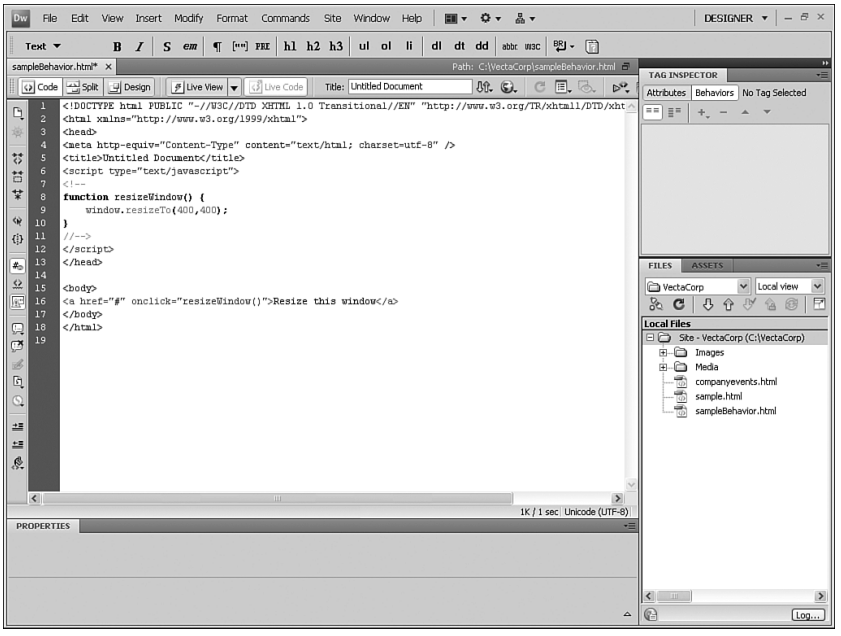

FIGURE B.6 Dreamweaver adds the JavaScript code necessary for the behavior to work.

Experiment with the functionality by saving your work and running it in the browser by pressing F12 (Option-F12). Clicking the link forces the browser to resize itself to 400 pixels high by 400 pixels wide.

The Advanced Resizer Behavior

Aside from the basics that we have covered so far, your behaviors can accept input from your users, just as the `SUP` object did. Up until this point, the behavior did no more than resize the browser window to the values that you provided to it. But what if you wanted to change it so that users could enter their own values? This could be accomplished by performing the following steps:

1. First, reopen the `windowResizer.html` file, if it's not open already, and change the UI to accept input from a standard HTML text box as follows:

```
<body>
<form name="resizeForm">
<table width="200">
<tr><td colspan="2">
What do you want to resize the browser window to:
</td></tr>
<tr><td>Width:</td><td>
<input type="text" name="width" size="10">
</td></tr>
```

```
<tr><td>Height:</td><td>
<input type="text" name="height" size="10">
</td></tr>
</table>
</form>
  </body>
```

2. Change the `applyBehavior()` function to accept the two text field values and return a concatenated value for width and height:

```
function applyBehavior() {
    var width = document.resizeForm.width.value;
    var height = document.resizeForm.height.value;
    return "resizeWindow(" + width + ", " + height + ")";
      }
```

3. Modify the defined function so that it accepts the two values as parameters: width and height.

```
function resizeWindow(width, height) {
    window.resizeTo(width, height);
  }
```

4. Save the behavior and click the Refresh button in the Behaviors panel (or restart Dreamweaver if you have to). Reopen `sampleBehavior.html` and delete the behavior that it currently has attached to it.
5. Reapply the Window Resizer behavior. This time, you are asked to type the values for the resizer. Figure B.7 shows the dialog box that appears.

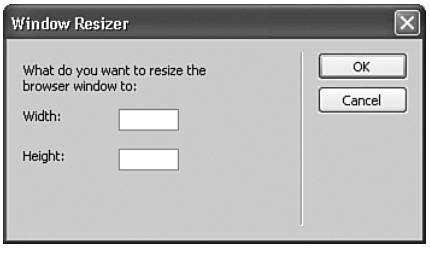

FIGURE B.7 The UI that you created should appear, allowing you to input values for the resizer.

Enter some numeric values and click OK to apply the behavior to the link. Save your work and test the results in the browser by pressing F12 (Option-F12). Clicking the link will cause the browser window to resize itself based on the values that we entered in Dreamweaver.

Advanced Behavior Functions

In addition to the basic elements that are required for the behavior to work, you can include optional functions to enhance the usability of your behaviors. Listed next are a few of them:

- **`initializeUI()` function**—Adding this function to the head of your behavior causes the cursor to land inside a text box that you specify. It also makes sure that the dialog box is selected when it is called from the panel list. You are free to do anything you want with this function, although the following code is typical for what you might use:

```
function initializeUI() {
     document.resizeForm.width.focus();
     document.resizeForm.width.select();
}
```

  For usability purposes it is always a good idea to add this function, especially when working with form elements within a behavior file.

NOTE

You will also need to add the `onLoad` event to the body tag for the function to be called: `<body onLoad="initializeUI()">`.

- **canAcceptBehavior() function**—Aside from enabling you to gray out specific events that you do not want the user to be able to use, you can also use this function to specify which event should be used as the default. The following function can be added to your script to default the event to `onMouseUp` rather than `onClick`:

```
function canAcceptBehavior() {
    return("onMouseUp");
}
```

- **inspectBehavior() function**—If you've been working with the newly created behavior, you will have noticed that when you double-click the behavior in the Behaviors panel, it doesn't remember the values you entered—instead it forces you to reenter the values. The `inspectBehavior()` function can be added to solve this problem:

```
function inspectBehavior(resizeFunctionCall) {
    var argArray = new Array;
    argArray = extractArgs(resizeFunctionCall);
    document.resizeForm.width.value = argArray[1];
```

```
document.resizeForm.height.value = argArray[2];
    }
```

Notice how the values of the text boxes in the UI are placed into an array. This is how Dreamweaver stores the values you have entered.

Just after the closing script tag, place a link to the shared `string.js` file. The `string.js` file contains functions that are needed for the behavior to function appropriately.

```
<script src="../../../Shared/MM/Scripts/CMN/string.js"></script>
```

TIP

The Configuration/Shared folder contains helpful scripts you can use when working with extensions. The `string.js` file is one of those scripts that comes in handy when working with strings.

Working with `.js` Files

If you look inside the Actions folder again, you will notice that your file is the only one without an accompanying `.js` file. This is done to separate the client logic from your UI, resulting in cleaner code that is easier to maintain. You, too, can maintain cleaner code by creating a separate `.js` file for your behaviors. To create a new `.js` file for your new behavior, follow these steps:

1. Cut all the script content out of the head of your `windowResizer.html` file.
2. Create a new file by selecting File, New.
3. Select the JavaScript option from the Basic Page category and click Create.
4. Paste your code in and delete the script tags.
5. Make sure to save the file in the same location as your HTML file, naming it **`windowResizer.js`**.
6. In the `windowResizer.html` file, add the following line:

   ```
   <script src="windowResizer.js"></script>
   ```

7. Save the file and refresh the Behaviors panel.

Your page works the same. The difference here lies in the fact that we've separated the code into its own `.js` file, making it easier to modify the code and user interface independently of one another.

Sharing Extensions Through the Adobe Exchange

After you create your first distributable object or behavior, you might want to share it with the world. Sharing extensions is not all that uncommon; in fact, Adobe has a whole section of its website devoted to extension sharing: The Adobe Exchange is made up of a

number of individual exchanges that are separated by product. This section focuses on preparing your extensions for submission to the Adobe Exchange, or more specifically the Dreamweaver Exchange, including details regarding the following:

- **Documentation**—Documentation means creating a help file so that everyone who is using your extension knows how it works.
- **Distribution**—Distribution means using the installer file to create an MXI file and bundling it into an extension using the Extension Manager.
- **Submission**—Submitting your extension to the Dreamweaver Exchange is the final step.

Documentation

A good extension is useful only if people know how to use it. You've probably noticed by now that when you select a preinstalled behavior that contains a dialog box, you're either given a textual explanation toward the bottom of the text box or you're given a Help button. The Help button is used to allow the user of the behavior to access online help, which provides more in-depth analysis of what the behavior can do and how it can be used. To create documentation for your extension, follow these steps:

1. Create a new HTML file in Dreamweaver and write some information about how your extension works. You can also include a screen capture if you think that is necessary.
2. Create a folder called **`CustomBehaviors`** within the Configuration\Shared folder. Save the help file into this folder and call it **`windowResizer_Help.html`**.
3. Open the `WindowResizer.js` file and create a new function called **`displayHelp()`**. This function is also part of the extension API:

```
function displayHelp() {
var theURL = dw.getConfigurationPath();
theURL += "/Shared/CustomBehaviors/windowResizer_Help.html";
dw.browseDocument(theURL);
  }
```

   Notice that the code makes calls to other methods in the API, specifically `getConfigurationPath()` and `browseDocument()`. `getConfigurationPath()` dynamically maps a path to the user's configuration folder. After the folder has been found, you append the path to the location where the file should reside. Then call the `browseDocument()` method, which opens the file that resides in that path.
4. Save that file and restart Dreamweaver, or Refresh the Behaviors panel if that option is available to you. Reopen the `sampleBehavior.html` file, select the link, and remove the existing behavior. Now reattach the behavior by selecting the Window Resizer option from the Custom Behaviors submenu located within the behaviors list. You'll notice that this time the Window Resizer dialog box contains a Help button, similar to Figure B.8.

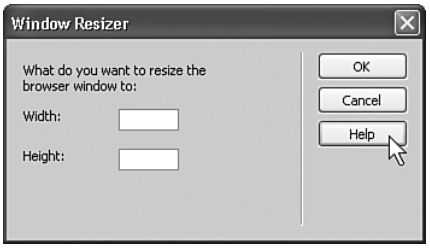

FIGURE B.8 A browser window is launched complete with your help file.

Clicking the Help button launches your custom help page (`windowResizer_Help.html`).

Distribution

After you've completely documented how your behavior should work, you are ready to package it up for distribution. Using Adobe Extension Manager CS4, shown in Figure B.9, you can package your extension into a special installation file known as a Macromedia Extension Installation, or an MXI.

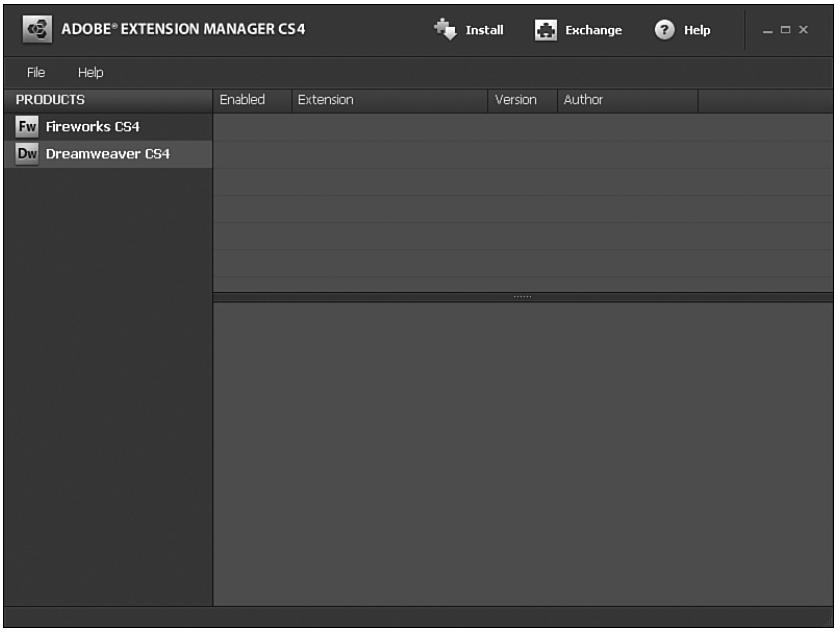

FIGURE B.9 The Extension Manager enables you to package your extension into a special installation file known as an MXI.

The process of creating this MXI file is easier than you might think and can be accomplished by following these steps:

1. Put all the necessary files (help file, HTML file, JS file, and GIF file [if packaging an object]) into a separate folder outside of the Configuration folder. I usually create a new folder at the root of my hard drive directory called **`Package`** and place all my files into that folder for packaging into a MXI file.
2. Copy the `Sample.mxi` file from the Extension Manager's Samples directory (`C:\Program Files\Adobe\Adobe Extension Manager CS4\Samples\Dreamweaver\` or `/Applications/Adobe Extension Manager/Samples/Dreamweaver/`) into the directory that you've just created and rename it **`windowResizer.mxi`.** You can use this as a template for building your own installation file. The MXI file is basically an XML file that contains information, such as the platform the extension requires, the author's name, the type of extension, and a brief description.
3. Open the file in Dreamweaver and fill in all the required information for name, version, type, product name, author name, description, and filenames. Note that the filenames section requires a location from which to grab the files and a destination to which users install the files. The finished code should look similar to Figure B.10.

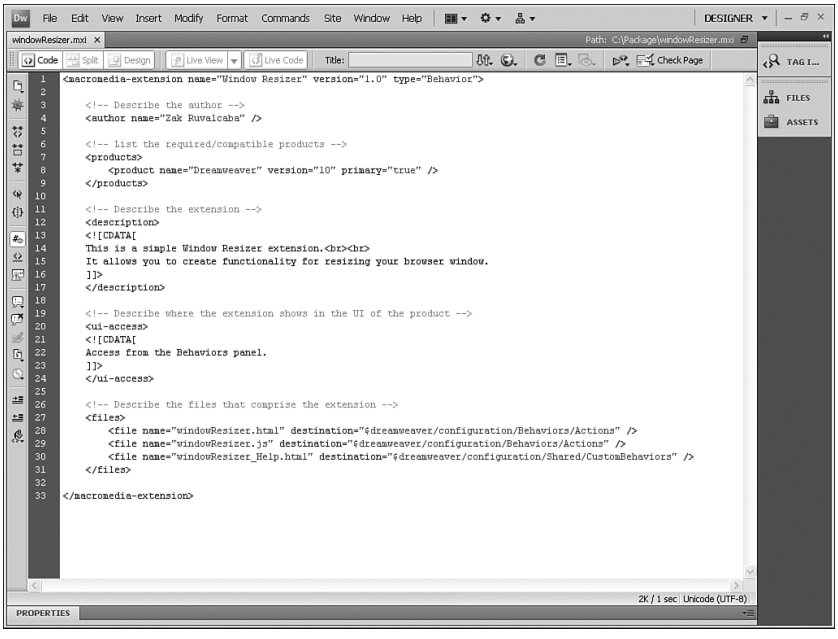

FIGURE B.10 Modify the MXI file to include all relevant information.

4. Open the Extension Manager. With the Extension Manager open, choose File, Package Extension. Locate the MXI file and select OK (Open). The Save Extension

Package As dialog box appears. Save the file as **`windowResizer.mxp`** and click OK (Save). The MXP (Macromedia Extension Package) file appears in the same directory as the MXI file.

To install the extension, follow these directions:

1. Open the Extension Manager if it's not already open.
2. Select File, Install Extension. Optionally, you can double-click the MXP file to install the extension. The Adobe Extension Manager license agreement dialog box appears. Click Accept and then click Yes. The extension then installs.

The package is installed into the appropriate folders and the extension appears in the Extension Manager, as shown in Figure B.11.

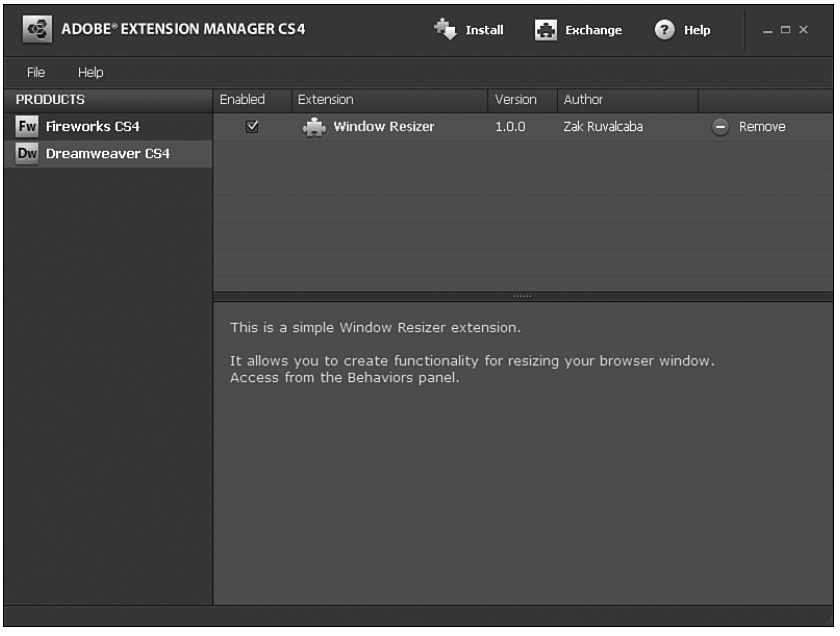

FIGURE B.11 The extension appears in the Extension Manager.

To begin using the extension, you must restart Dreamweaver.

Submission

After you have completely documented, tested, and packaged your extension, you are ready to share it with the rest of the development community on the Dreamweaver Exchange. The process is relatively simple and can be completed by following these steps:

1. Visit the Dreamweaver Exchange by first navigating to the main Adobe Exchange at the following URL: www.adobe.com/cfusion/exchange/. Select the Dreamweaver link

from the Exchange product section in the middle of the page. The Dreamweaver Exchange will appear, similar to Figure B.12.

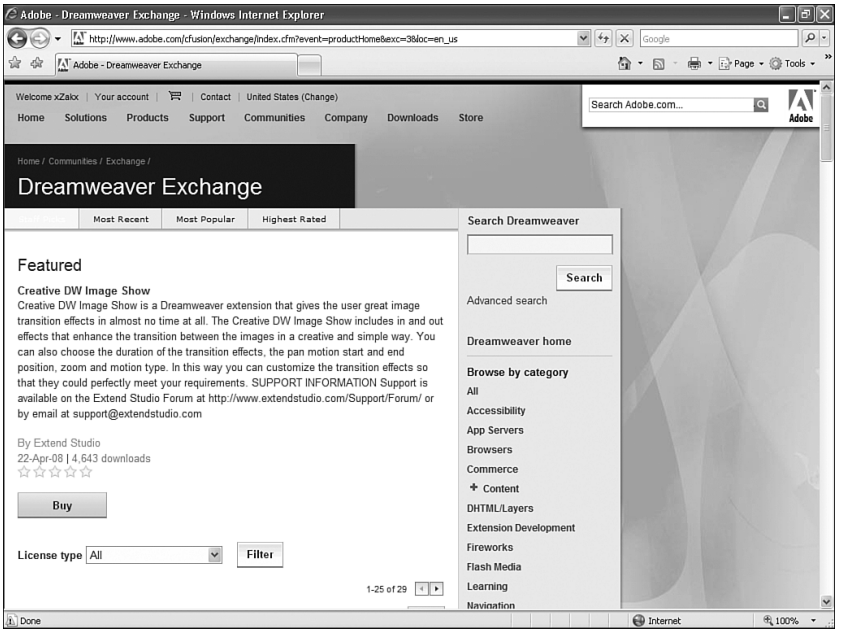

FIGURE B.12 The Dreamweaver Exchange site.

2. Click the Upload to the Exchange link from the options available to you on the bottom right.
3. Create an account if you do not have one already. If you do have one, log in now.
4. After you've logged in, click the Begin Upload Process button and follow the onscreen prompts until you are instructed to upload your extension package. After this screen appears, browse to the MXP file that you created in the previous section and click Next to proceed to the Upload Details screen.
5. Within the Upload Details screen, fill out all the information about your extension. This is where you get to tell Adobe what your extension does, what products it works with, and so on. When you've filled in this information, click Next to proceed to the Author Confirmation page. Click Next to upload the extension to the Exchange. You might also click the View Your Upload button to view any and all extensions that you've uploaded to the Exchange.

From this point forward, Adobe reviews your extension. When it's approved, the extension appears in the Adobe Exchange, under the product category you submitted it for, and anyone in the world can download and use it.

The Server Behavior Builder

Dreamweaver provides server behaviors to do the most common server-side processing tasks with the most popular application servers. Getting your web page to interact with a database is easily done with Dreamweaver, and most of the options you would want are covered. However, there are bound to be occasions when you want to do something but no server behavior is available for you to do it. When this happens, you have a number of options:

- You can go to the Dreamweaver Exchange and download a server behavior to create the functionality you want.
- You can hand code the file to include the server-side code that's needed to do the job.
- You can build your own server behavior so that the next time you want to do the same job, the code is already there for you or other members of your collaborative team.

In the previous section, we discussed extensions in some detail, but concentrated on extensions that used client-side JavaScript. Server-side behaviors work in almost the same way. In this section, we won't repeat the concepts learned in the previous sections; instead, we'll look at building your own server-side behaviors from scratch by using the Server Behavior Builder.

Server Behaviors

The built-in server behaviors are powerful because they allow nonprogrammers or those with only a limited knowledge of server-side programming to perform common functions such as displaying database records. They can be used by web designers to visually design a page to produce simple web applications. If they need to modify the page after design—to reorder database columns or to add in more data, for example—there is no need to hunt through and modify code; the server behavior can be easily changed.

Server behaviors are also useful to experienced programmers because much of web application coding is the repetition of common tasks, such as creating a recordset, opening it, and displaying the records. Server behaviors let you automate these common tasks with robust standard code so that you can concentrate on the difficult parts.

Hand coding unique pages gets the job done, but unless you are a web design team of one, and you are sure that no one else will want to modify the page after you have finished with it, I would recommend adding a custom server behavior. Even if you are the sole developer, you can save yourself time in the long run by creating your own server behaviors. Let's create a simple server-side behavior now.

Writing a Simple Cookie Behavior

To demonstrate the process of writing custom server behaviors, we'll outline the process of manually writing a cookie using traditional ASP.

NOTE

Cookies are pieces of text that are generated by a server-side technology and then sent from the web server to the client (browser) in an effort to store information about the client that the server-side technology can later reuse. Generally, cookies are used for authenticating, tracking user sessions, and maintaining specific information about users, such as site preferences or content contained within a shopping cart. By default, cookies are stored in the user's temporary Internet files folder.

Cookies are sent by the server to the client along with the requested page. If the cookie is given an expiration date, the client's browser saves the cookie on the client's machine so that it can be referred to during future visits to the website. The client sends the cookie back to the server with each page request and form submission, and a server-side script can use this cookie value as part of the dynamic page-generation process. Although cookies are sent and processed by the server, you can also read and write cookies using JavaScript after the page has arrived at the client. Dreamweaver includes two code snippets to read and write cookies, using JavaScript, which are shown in Figure B.13.

FIGURE B.13 Dreamweaver includes two client-side behaviors for reading and writing cookies.

In the following sections, we'll create our own functionality for reading and writing cookies. Rather than using JavaScript, however, we'll use two custom server behaviors that

we'll write ourselves. Our first server behavior creates the form elements within the `<form>` tag, as well as the necessary ASP code above the opening `<html>` tag to write the cookie. We'll call that page **`writecookie.asp`**. Next, we'll create a second server behavior and add it to a page called **`readcookie.asp`**. This server behavior will effectively read the cookie information that was written from the first page and write the value within the second page.

To start writing the behavior, follow these steps:

1. Open a new ASP page by selecting File, New. Choose the ASP VBScript option from the Blank Page category, select the <none> option from the Layout list, and click Create.
2. With the new page open, add the text **`Create a New Cookie`** and assign it the Heading 3 format from the formatting drop-down menu in the Property inspector.
3. Select Window, Server Behaviors to open the Server Behaviors panel.
4. Click the Add (+) button and choose New Server Behavior, as shown in Figure B.14.

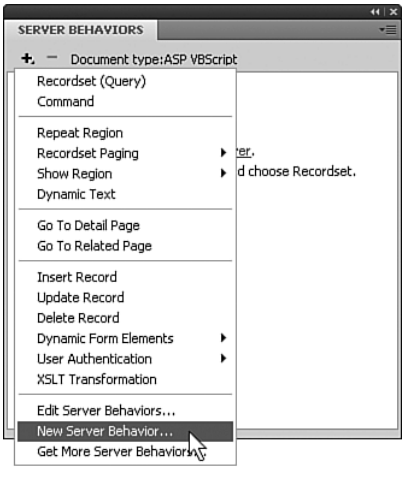

FIGURE B.14 Choose the New Server Behavior option from the Add Behavior menu to start writing your behavior.

5. You are presented with the New Server Behavior dialog box. Choose the document type you want to use (Dreamweaver should guess from the site you have open) and give the behavior a name. For this example, use **`Write Cookie`** (spaces are allowed here). Remember that other people are going to use the behavior, so make the name descriptive of its function. This is a completely new behavior, so do not enable the Copy Existing Server Behavior check box. The result looks similar to Figure B.15.
6. Click OK.

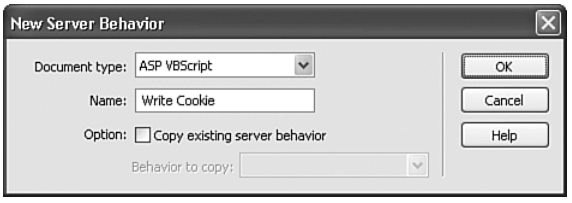

FIGURE B.15 Create a name for your new server behavior.

7. You should now be presented with the actual Server Behavior Builder dialog box. This is where most of the work is done. You write blocks of code here and tell Dreamweaver where you want the code to be written on the page. Begin by adding a code block by clicking the Add (+) button on the top left. The Create a New Code Block dialog box appears. Accept the suggested name by clicking OK.
8. The Code Block window asks you to replace the text with your own code. Let's go ahead and do just that. We can start by adding the code that will make up the interface:

```
<form name="form1" method="post" action="writeCookie.asp">
User ID:<br>
<input name="txtUserID" type="text" id="txtUserID"><br>
Username:<br>
<input name="txtUsername" type="text" id="txtUsername"><br><br>
<input type="submit" name="Submit" value="Submit">
  </form>
```

This code block adds a new form to the page complete with two text boxes and a Submit button. The result of adding the code looks similar to Figure B.16.

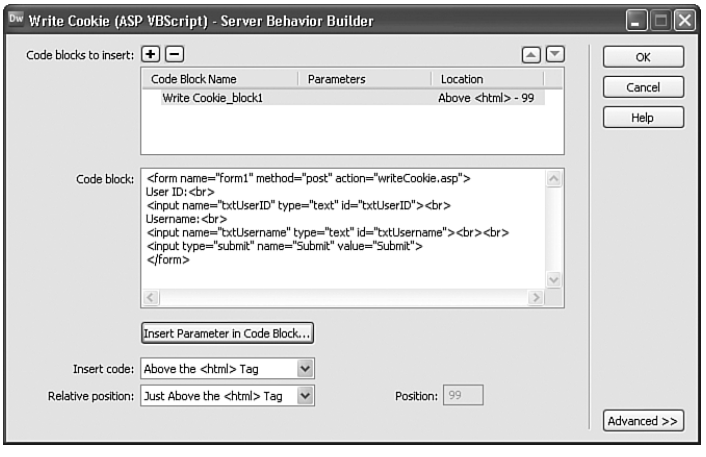

FIGURE B.16 Add the HTML that will make up the interface.

9. Our next task is to make changes so that the server behavior knows to insert the new form and its contents within the `<body>` tag, but just underneath a new heading

that we'll be creating. You can do this by selecting the Relative to a Specific Tag option from the Insert Code menu. Next, select the `h3` tag from the Tag drop-down menu. Finally, select the After the Closing Tag option from the Relative Position drop-down menu. The result resembles Figure B.17.

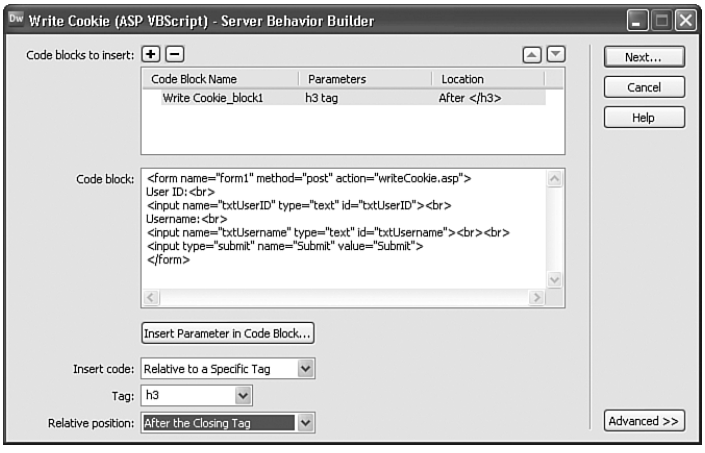

FIGURE B.17 Customize the positioning of the code block.

10. Let's add another code block so that we can add the necessary ASP code above the opening `<HTML>` tag. Click the Add (+) button one more time. Again, accept the default value and click OK. In the Code Block text box, add the following code:

```
<%
If (Request.Form("Submit") = "Submit") Then
Response.Cookies("WriteCookie")("Key") = "1"
Response.Cookies("WriteCookie")("UserID") = Request.Form("txtUserID")
Response.Cookies("WriteCookie")("Username") = Request.Form("txtUsername")
Response.Cookies("WriteCookie").Expires = Date + 2
Response.Redirect("readCookie.asp")
End If
  %>
```

This code effectively checks whether the form is being posted back to itself. If it is, it creates a new cookie called `WriteCookie`, sets a key value equal to 1, sets a second key called `UserID` equal to the value of the txtUserID text box, sets a third key called `Username` equal to the value of the txtUsername text box, sets the cookie expiration to expire two days from the time the cookie is created, and then redirects the user to `readCookie.asp`. The result of the addition resembles Figure B.18.

11. Set the position of where the code block is to be inserted. By default, Dreamweaver has the value Above the `<HTML>` tag already selected for you. You can stick with this option. Now that you are done building your new server behavior, click Next. The Generate Server Behavior dialog box appears. Click OK.

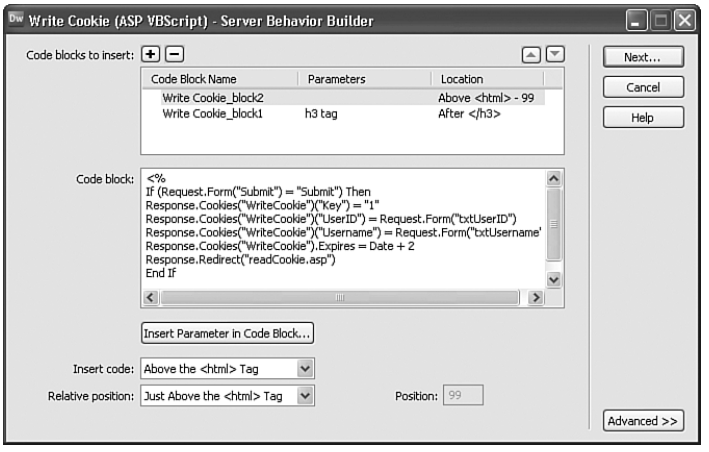

FIGURE B.18 Add the ASP code that creates the cookie and sets the key values.

You are now ready to add your new server behavior to the page. To select the new server behavior, click the Add (+) button from the Server Behavior tab in the Application panel. Your new server behavior appears, as shown in Figure B.19.

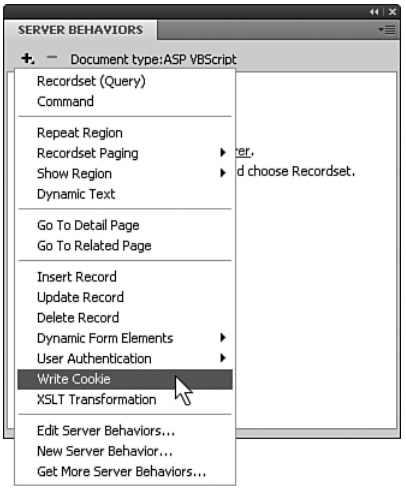

FIGURE B.19 The new server behavior appears in the server behavior list.

Select your new server behavior. The Write Cookie dialog box appears, similar to Figure B.20.

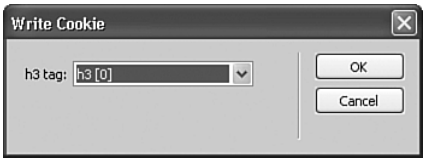

FIGURE B.20 The Write Cookie dialog box appears.

Because you are selecting the placement for a portion of the code just after the <h3> tag, this dialog allows you to specify which <h3> tag to place the code after, assuming that you had more than one. Because there is only one <h3> tag on our page, click OK. The form elements should be inserted into the page. Notice that the designer displays the page based on the code you built in the Server Behavior Builder, similar to Figure B.21.

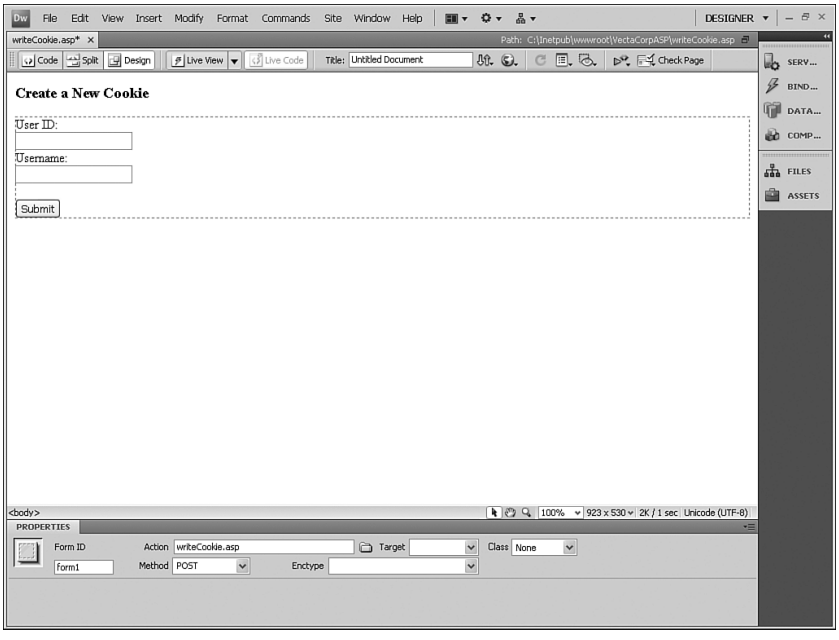

FIGURE B.21 The designer shows the form based on the code you added within the Server Behavior Builder.

NOTE

Because we wrote our behavior to be relevant to the <h3> tag, running this behavior on a page that excludes the <h3> tag will result in an error.

Save your file as `writeCookie.asp`. Before we test the functionality, however, let's build another server behavior to read the existing cookie. Start by creating a new page by

selecting File, New. Select the ASP VBScript option from the Blank Page category, choose the <none> option from the Layout list, and click Create. Again, choose New Server Behavior from the Add (+) button in the Server Behaviors tab of the Application panel. The New Server Behavior dialog box appears. Give your new behavior the name **`Read Cookie`** and click OK.

Click the Add (+) button in the upper left to add a new code block, accept the default value, and click OK. Then add the following code to the Code Block text box:

```
<%
If (Request.Cookies("WriteCookie")("Key") = "1") Then
     Response.Write(Request.Cookies("WriteCookie")("UserID"))
     Response.Write(Request.Cookies("WriteCookie")("Username"))
End If
%>
```

This code initially checks to make sure that a cookie named `WriteCookie` has a key with a value of 1. If it does, it writes both values of the UserID key and the Username key to the page by using the `Response.Write()` method. The result of adding the code to the Code Contents window should resemble Figure B.22.

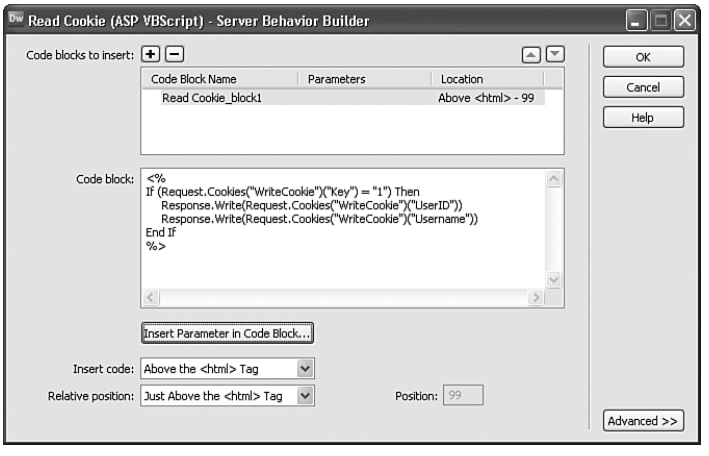

FIGURE B.22 Add the code that reads the values of the keys in the cookie.

Click OK. Now select the Add (+) button from the Server Behavior tab in the Application panel. Your new Read Cookie server behavior appears. Select it. Switching to Code view reveals the newly inserted server behavior code. Save your work as **`readCookie.asp`**. Now open `writeCookie.asp` in the browser. Add a user ID and username to the text boxes and click Submit. You'll be instantly redirected to `readCookie.asp`, where the values will be shown in the browser window.

To view the newly created cookie, in Internet Explorer, select Tools, Internet Options. From the General tab, select Settings within the Browsing History pane (if you're using

IE 7). When the Settings dialog box appears, select View Files. The Temporary Internet Files window appears, complete with your newly generated cookie.

Summary

As you've seen, customization in Dreamweaver is a nice way of managing your workflow and customizing your development environment. Although you have only scratched the surface by working through this appendix, you can begin to see the endless possibilities available to you by developing your own objects and behaviors. The Dreamweaver Exchange provides Dreamweaver users with features the software industry has seldom seen before: third-party software in the form of modular code. Dreamweaver opens the door to a whole new world of development—one that is not limited by design potential or development knowledge but by your willingness to take the application as far as your workflow needs require. The Adobe Exchange is full of extensions that vary in cost from free to potentially hundreds of dollars, depending on what they do. Something to remember is that Dreamweaver is professional web development software, and many third-party extensions can be used over and over and from site to site. That's flexibility and power that can be well worth paying for.