Chapter 4

XML Transference from Data to Layout

The speed at which XML was adopted across the business sector was stunning. My first indication of the XML phenomenon came when I was teaching a seminar on maximizing the potential of Dreamweaver 3 at the Seybold conference in Boston. Trying to gauge the audience's level of expertise, I asked how many people had used Dreamweaver before. All but two or three raised their hands. When I asked how many people were familiar with Cascading Style Sheets, only about 1/3 responded. "How many people here use XML in their business?," I asked next. Three-quarters of the audience—an audience of Web developers, mind you—shot up their hands. Clearly, XML was a technology about to explode.

Dreamweaver has long maintained—since version 3, in fact—a strong XML connection. Importing and exporting of the well-formed XML file is only a menu option away. Of course, Dreamweaver assumes you know how to format the XML file properly for importing and how to make the most of exported data. Don't worry if you don't know how to do these things; those topics are part of what will be covered in this chapter.

The standard Dreamweaver XML features are powerful ones, but really only serve as a foundation for what is possible. In this chapter, you'll see how to automate the production of Web pages from XML data. We'll also explore techniques for structuring XML data as content, ready for import into a

Dreamweaver template. Finally, we'll look at how to extract the content from a document derived from a template and how to store the information in a data source.

A Brief Introduction to XML

XML, short for Extensible Markup Language, has often been described as a customizable version of HTML. Although this depiction is accurate to a degree, it doesn't really go far enough to distance it from HTML and characterize the language's strengths. To me, XML is pure structure. Each XML tag is only present to contribute to the structure of a document. Better still, the very name of each XML tag describes the content it contains, furthering the structural integrity of the document.

XML files begin with a statement that declares the XML version used. By default, Dreamweaver MX creates XML version 1.0 documents that specify the encoding:

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

Internally, XML syntax is similar to HTML with a few key differences:

- Empty XML elements or tags include a closing slash, like this:

```
<bookImage src="jloweryImage.jpg" width="150" height="150" />
```

- The values of attributes must be quoted.
- Standalone attributes are not permitted. That is, `checked` as an attribute is not permitted, but `checked="true"` is.
- To avoid processing tags within text data, XML uses the `CDATA` element. `CDATA` stands for *character data*, and it is designated by surrounding the text with `<![CDATA[` and `]]>`. Here's an example:

```
<![CDATA[To designate a table of contents item, enclose the entry
with a <toc>...</toc> tag pair.]]>
```

Structurally, XML documents tend to be made up of multiple sets of tags following the same format. For example, if I were to describe a series of books I've read, the basic form might look like this:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<books>
  <book>
    <bookTitle name="">
    <authorName name="">
    <bookDescription>
      <![CDATA[]]>
    </bookDescription>
  <book>
</books>
```

With a number of entries completed, the XML file would look like this:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<books>
  <book>
    <bookTitle name="Critical Space">
    <authorName name="Greg Rucka">
    <bookDescription>
      <![CDATA[Bodyguard Atticus Kodiak is hired by someone who
      attempted to once kill him.]]>
    </bookDescription>
  <book>
  <book>
    <bookTitle name="Blackwater Sound">
    <authorName name="James W. Hall">
    <bookDescription>
      <![CDATA[Thorn abandons his role as Florida fisherman to
      stop the injustice brought by the rich and powerful
      Brasswell family.]]>
    </bookDescription>
  <book>
```

First of three children

continues

```
  <book>
    <bookTitle name="Pursuit">
    <authorName name="Thomas Perry">
    <bookDescription>
      <![CDATA[Who is the hunter and who is the hunted in
      this book of criminologist vs. serial killer?]]>
    </bookDescription>
  <book>
</books>
```

In this example, the overall structural element is the `<books>` tag, which has three nodes or *children* in the `<book>` tags. Within each `<book>` child, the same descriptive tags are used with varying values. We'll see this exact type of format when we examine XML documents that are exported from Dreamweaver.

Exporting Template Content to XML

Most of Dreamweaver's XML focus is dedicated to working with templates and template-derived documents. Dreamweaver regards the locked areas of a template as the presentation layer of a document and the content within the editable and repeating regions as well as some other template markup as the data. Dreamweaver can only pull XML data from an instance of a template and, similarly, import XML data into a template instance.

Exporting a Single Document

As noted earlier, Dreamweaver provides a direct menu command for extracting the XML content from templates: File, Export, Template Data as XML. When invoking this command—which becomes active only when the current document is template derived—the Export Template Data as XML dialog box is displayed (see Figure 4.1). For basic templates—ones that use only editable regions—Dreamweaver can format the XML output in one of two ways. The standard Dreamweaver XML approach lists every region as an `<item>` tag, identified separately by the `name` attribute. For example, if a template had three editable regions—such as `bookTitle`, `authorName`, and `bookDescription`—then selecting the Use Standard Dreamweaver XML Tags option would output the following:

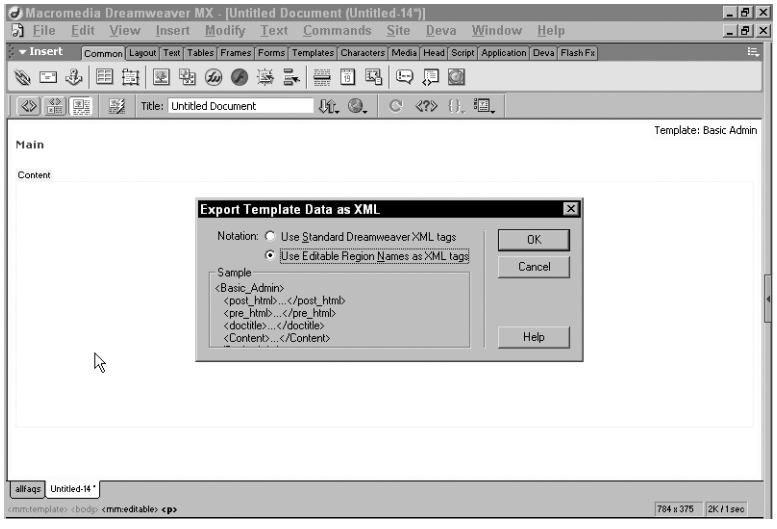
NOTE: You might find it helpful to examine output from Dreamweaver's Export Template Data as XML command; you'll find examples in Listing 4-1 (standard items sample) and Listing 4-2 (editable regions sample) at the end of this section.

```
<item name="bookTitle"><![CDATA[Critical Space]]></item>
<item name="authorName"><![CDATA[Greg Rucka]]></item>
<item name="bookDescription>><![CDATA[Bodyguard Atticus Kodiak is
hired by someone who attempted to once kill him.]]></item>
```

In contrast, the Use Editable Region Names as XML Tags option formats the data like this:

```
    <bookTitle>![CDATA[Critical Space]]></bookTitle>
    <authorName><![CDATA[Greg Rucka]]></authorName>
    <bookDescription><![CDATA[Bodyguard Atticus Kodiak is hired by
someone who attempted to once kill him.]]></bookDescription>
```

Note constant use of CDATA



4.1

The Export Template Data as XML dialog box offers two export styles—as long as the only template markup tags in the document are editable regions.

There are other differences as well. For instance, the syntax for the master template under the standard Dreamweaver XML tags is this:

```
<templateItems template="/Templates/BasicBookList.dwt"
codeOutsideHTMLIsLocked="false">
...
</templateItems>
```

as opposed to this:

```
<BasicBookList template="/Templates/BasicBookList.dwt"
codeOutsideHTMLIsLocked="false">
...
</BasicBookList>
```

As noted, complex template documents—those that have repeating regions or template parameters as well as editable regions—can only be output in the standard Dreamweaver format. Several additional tags are used to note the enhanced template markup. A template parameter—the only evidence of a conditional region—looks like this:

```
<parameter name="dbOnSale" type="boolean"
passthrough="false"><![CDATA[true]]></parameter>
```

A repeating region is coded this way:

```
<repeat name="RepeatRegion1">
  <repeatEntry>
    <item name="BookTitle"><![CDATA[Critical Space]]></item>
    <item name="AuthorName"><![CDATA[Greg Rucka]]></item>
    <item name="MainCharacter"><![CDATA[Atticus Kodiak]]></item>
  </repeatEntry>
  <repeatEntry>
    <item name="BookTitle"><![CDATA[Blackwater Sound]]></item>
    <item name="AuthorName"><![CDATA[James W. Hall]]></item>
    <item name="MainCharacter"><![CDATA[Thorn]]></item>
  </repeatEntry>
</repeat>
```

That's it. The following listings show the completed code.

NOTE: Don't get the idea that template data can only be exported from static pages. Dynamic pages, which often include code outside of the `<html>` tag pair, can also be exported. Dreamweaver uses a special syntax for code that appears before the opening `<html>` tag: `<item name="(code before HTML tag)">` or `<pre_html>`. Similar tags are used for code that appears after the closing `</html>` tag.

NOTE: If your template region names include special characters—including spaces or underscores—then Dreamweaver only lets you export using the standard Dreamweaver method.

Listing 4-1 **Standard Items Sample** (04_standarditems.xml)

```
<?xml version="1.0"?>
<templateItems template="/Templates/Basic Admin.dwt"
codeOutsideHTMLIsLocked="false">
   <item name="(code after HTML tag)"><![CDATA[
]]></item>
   <item name="(code before HTML
tag)"><![CDATA[<%@LANGUAGE="VBSCRIPT"%>
]]></item>
   <item name="doctitle"><![CDATA[
<title>Untitled Document</title>
]]></item>
   <item name="Content"><![CDATA[
<p>The content goes here.</p>
<p> </p>
]]></item>
</templateItems>
```

Listing 4-2 **Editable Regions Sample** (04_editableregions.xml)

```
<?xml version="1.0"?>
<Basic_Admin template="/Templates/Basic Admin.dwt"
codeOutsideHTMLIsLocked="false">
   <post_html><![CDATA[]]></post_html>
   <pre_html><![CDATA[<%@LANGUAGE="VBSCRIPT"%>
</pre_html>
   <doctitle><![CDATA[
<title>Untitled Document</title>
]]></doctitle>
   <Content><![CDATA[
<p>The content goes here.</p>
<p> </p>
]]></Content>
</Basic_Admin>
```

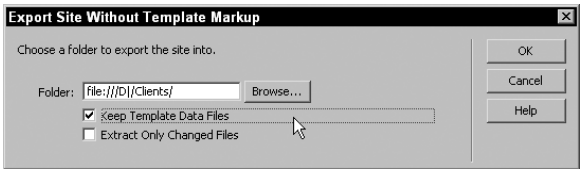
Exporting an Entire Site

Undoubtedly, the Export, Template Data as XML command is quite useful at extracting the content from a templated page; unfortunately, it's also somewhat tedious. If you are responsible for exporting the content from an entire site, you have quite a repetitive task ahead of you when you're using just this feature. Luckily, Dreamweaver includes an equally powerful command for extracting all the data from all the template-derived pages in a site.

Although the command for exporting a single page is front and center, you really have to dig to find the equivalent site-wide command. In fact, you have to perform an entirely different—somewhat antithetical—operation to get the XML output. Dreamweaver MX includes the ability to export a site, completely stripping out all the template markup from template-derived documents. When you choose Modify, Templates, Export Without Markup, the dialog box shown in Figure 4.2 is displayed.

4.2

To extract all the data as XML from a site, you first must choose Templates > Export Without Markup.



NOTE: To avoid having to ask for individual names for each file exported, Dreamweaver automatically appends the .xml extension to whatever the original filename is—including the extension. For example, marchbooklist.htm becomes marchbooklist.htm.xml.

To enable the XML operation, make sure that the Keep Template Data Files option is selected. This is the signal to Dreamweaver to make two copies of template-derived documents: one without template markup and another in XML data format. Dreamweaver stores both files in the same folder. If you've previously exported the site and want to update the data files, select Extract Only Changed Files.

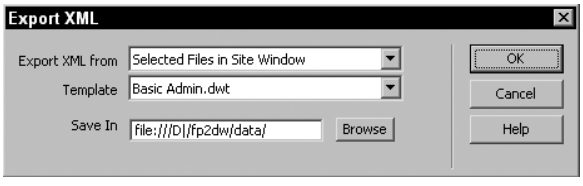
Exporting Selected Files in a Site

So far, we've seen how Dreamweaver can export XML data from a single page or from an entire site. However, what if you need something in between? What if you need to export the five files you're working on or only a couple of folders of files? What if you don't want to make a template-less copy of the site to get the exported files?

Although Dreamweaver makes exporting template data as XML programmatically appear straightforward, there's an aptly named function,

`dw.exportTemplateDataAsXML()`. It's not as easy as it seems. Every time the `exportTemplateDataAsXML()` function is called, the Export Template Data as XML dialog box opens, disrupting the automatic nature of the operation. Luckily, Dreamweaver makes it fairly easy to get an array of all the editable regions in a document with another API function, `dom.getEditableRegionList()`. That function serves as the basis for the Export XML extension.

The Export XML extension (see Figure 4.3) allows the user to select the scope of the operation (Current Document, All Open Documents, Selected Files in Site Panel or Entire Site), which template to declare in the XML file, and where to store the files.



4.3

Export XML uses Massimo Foti's site-wide library to work with a wide range of file selections.

Another real challenge is developing a system to supply the file URLs as needed, whether it's just from the current document, from the selected files, or from the entire site. Although Macromedia hasn't seen fit to provide this functionality, a robust solution has emerged from Dreamweaver's premier third-party developer, Massimo Foti. Fairly early on in the development of his extensions, Massimo came across this very problem: How do you process a single command across a site?

Over the years, Massimo has refined his library numerous times, and it is quite full featured and extremely useful. Although the code is too lengthy to reproduce and analyze in this chapter, I have included it as Bonus Listing 4-A (Massimo Foti's Site-Wide Library [04_siteutils.js]) on the book's web site, with Massimo's permission, as well as a small extension to demonstrate its use, shown here in listing 4-3.

TIP: To see the full range of Massimo Foti's work, visit `www.massimocorner.com`.

Listing 4-3 **Export XML** (04_exportxml.mxp)

```
<html>
<head>
<title>Export XML</title>
```

continues

```
<script language="javascript"
src="../Shared/Beyond/Scripts/Beyond_Site.js"></script>
<script language="JavaScript" src="../Shared/MM/Scripts/CMN/UI.js"
type="text/JavaScript"></script>
<script language='javascript'>

//init globals ***************************************************
var theSelect = findObject('fileList');
var theWildText = findObject('wildText');
var theTemplateList = findObject("template_list");
var theXML = "";


//****************** Primary Functions *************************

function commandButtons(){
    return new Array( 'OK', 'doCommand()', 'Cancel',
    'window.close()', 'Help', 'getHelp(theHelpFile)')
}
function exportXML(theURL) {
   var theDOM = dw.getDocumentDOM();
   if(theDOM.documentElement.innerHTML.indexOf("InstanceBegin
   template=") != -1) {
      var theERs = theDOM.getEditableRegionList();
      var theName, theData
      theXML = "";
      for (i = 0; i < theERs.length; ++i) {
         theName = theERs[i].getAttribute("name");
         theData = theERs[i].innerHTML;
         theXML += '<item name="' + theName + '"><![CDATA[' +
         theData + ']]></item>' + '\n';
      }
      var theXMLHeader = "";
      theTemplate = theTemplateList.options
      [theTemplateList.selectedIndex].text;
      var theTemplateFile = dw.getSiteRoot() + "Templates/" +
      theTemplate;
```

```
        theXMLHeader += '<?xml version="1.0"?>' + '\n';
        theXMLHeader += '<templateItems template="/Templates/' +
        theTemplate + '" codeOutsideHTMLIsLocked="false">' + '\n';
        theXML = theXMLHeader + theXML + '\n' + '</templateItems>'
        + '\n';
        var fileURL = findObject("folder_text").value +
        theURL.substr(theURL.lastIndexOf("/") + 1) + ".xml";
        r = DWfile.write(fileURL, theXML);
    }
    dw.releaseDocument(theDOM);
}

function doCommand() {
    var selectArray = new Array("currentDoc","openedDocs",
    "siteSelected","wholeSite");
    var theRes = 1
    var theWildCards = ".htm;.html;.shtm;.shtml;.asp;.cfm;.cfml;
    .php;.php3"

    for (var i=0; i<theSelect.options.length; i++){
        if (theSelect.options[i].selected){
            whichFiles = selectArray[i];
        }
    }
    switch (whichFiles){

        case "currentDoc":
            urlArray = getCurrentDoc();
            if(urlArray){
                exportXML(urlArray);
            }
            break;

        case "openedDocs":
            agree = confirm("This command cannot be undone.
            Proceed?");
```

continues

```
//If it's ok, go
if (agree){
   openFilesArray = new Array();
   //Get the currently opened files
   openFilesArray = getOpenedDocs();
   //Filter them to get just the ones matching extensions
   urlArray = filterFiles(theWildCards,openFilesArray);
   for (var i=0; i<openFilesArray.length; i++){
      exportXML(urlArray[i]);
   }
} else {
   return;
}
break;

case "siteSelected":
 var siteFocus,agree;
 siteSelectedArray = new Array();
 writeSiteSelectedArray = new Array();
 siteFocus = site.getFocus();
 if(siteFocus == "local" || "site map"){
    //Ask the user
    agree = confirm("This command cannot be undone.
    Proceed?");
    //If it's ok, go
    if (agree){
       //Get the urls of the files selected inside
       //the site window
        siteSelectedArray = site.getSelection();
       if (siteSelectedArray.length == 0 ||
       siteSelectedArray[0].indexOf(".") == -1) {
          alert("No files in Site window selected.
          \nChoose another Generate From option.")
          return;
       }
       //Filter them to get just the matching extensions
```

```
            urlArray = filterFiles(theWildCards,
            siteSelectedArray);
            for (var i=0; i<urlArray.length; i++){
               exportXML(urlArray[i]);
            }
         } else {
            return;
         }
      }
      else{
         alert("This command can affect only local files");
      }
      break;

   case "wholeSite":
      var agree;
      wholeSiteArray = new Array();
      writeFilesArray = new Array();
      //Ask the user
      agree = confirm("This command cannot be undone.
      Proceed?");
      //If it's ok, go
      if (agree){
         //Get all the urls of files in the site with
         //matching extensions
         wholeSiteArray = getWholeSite();
         //Filter them to get just the matching extensions
         urlArray = filterFiles(theWildCards,wholeSiteArray);
         for (var i=0; i<urlArray.length; i++){
            exportXML(urlArray[i]);
         }
      } else {
         return;
      }
      break;
}
window.close();
```

continues

```
    return;
}

function findFolder() {
    findObject("folder_text").value = dw.browseForFolderURL();
}

function getTemplateList() {
    //returns a list of templates in site
    var theTemplateDir = dw.getSiteRoot() + "Templates/";
    var theTemplates = new Array();
    theTemplates = DWfile.listFolder(theTemplateDir + "*.dwt",
    "files");
    if (theTemplates){
        loadSelectList(theTemplateList,theTemplates);
    }
}

function initUI() {
    getTemplateList();
}

</script>
</head>

<body onLoad="initUI()">
<form name="theForm">
  <table border="0">
     <tr>
       <td nowrap> <div align="right">Export XML from</div></td>
       <td nowrap> <select name="fileList" style="width:220px">
            <option selected>Current Document</option>
            <option>All Open Documents</option>
            <option>Selected Files in Site Window </option>
            <option>Entire Site</option>
         </select> </td>
```

```
    </tr>
    <tr>
      <td nowrap><div align="right">Template</div></td>
      <td nowrap><select name="template_list" id="template_list"
      style="width:220">
          <option selected>Loading templates.............</option>
        </select></td>
    </tr>
    <tr>
      <td><div align="right">Save In</div></td>
      <td><input name="folder_text" type="text" id="folder_text"
      style="width:155">
        <input type="button" name="Button" value="Browse"
        onClick="findFolder()"></td>
    </tr>
  </table>
</form></body>
</html>
```

After Massimo's function does the heavy lifting of finding all the required file URLs, that information is passed to the `exportXML()` function in the extension. As is often the case, the Document Object Model (DOM) for the document is first appropriated and put into a variable. Then the function tests to make sure that the document is derived from a template and that it's possible to export XML from it. Again, there is a Macromedia API function intended for this purpose—and again, we can't use it because it requires the document to be open before it will work. Because I don't want to open and close every document, I found another way to determine whether the file is template derived:

```
if(theDOM.documentElement.innerHTML.indexOf("InstanceBegin
template=") != -1)
```

This code walks down the DOM a bit and looks for the key words that indicate the document is a template instance. If so, we're ready for the process to begin by getting all the editable regions in the file:

```
var theERs = theDOM.getEditableRegionList();
```

The next significant action takes place in a loop where the editable region name and innerHTML are extracted and inserted into an XML format:

```
for (i = 0; i < theERs.length; ++i) {
   theName = theERs[i].getAttribute("name");
   theData = theERs[i].innerHTML;
   theXML += '<item name="' + theName + '"><![CDATA[' + theData +
']]></item>' + '\n';
}
```

Added to format output

Next, we're ready to set up the template name variable, which we'll soon insert into the XML file:

```
theTemplate = theTemplateList.options[theTemplateList.
selectedIndex].text;
```

The header for the XML file is constructed next, integrating the template name:

```
theXMLHeader += '<?xml version="1.0"?>' + '\n';
theXMLHeader += '<templateItems template="/Templates/' +
theTemplate + '" codeOutsideHTMLIsLocked="false">' + '\n';
```

Now the entire XML file is concatenated into one string:

```
theXML = theXMLHeader + theXML + '\n' + '</templateItems>' + '\n';
```

Closing XML tag

After building the file URL to store the XML file, the DWfile API is used to write it out:

```
var fileURL = findObject("folder_text").value +
theURL.substr(theURL.lastIndexOf("/") + 1) + ".xml";
r = DWfile.write(fileURL, theXML);
```

The final instruction in the code is used to release the memory used to work with the DOM—a necessary step when possibly processing an entire site:

```
dw.releaseDocument(theDOM);
```

Manually Importing from XML

On the flip side of exporting XML data from templates, Dreamweaver has the capability to import XML data into a template to create a new template-derived document. For the import operation to work as intended, the XML must be in a specific format. Each XML tag corresponds to a template region or markup. Both XML syntax used during export—the standard Dreamweaver `<item>` syntax and the editable region names as XML tags—are supported for import, with the same restrictions. For complex templates, including any template markup other than just editable regions, the standard Dreamweaver syntax must be used.

Perhaps the best way to understand the format required for import is to examine an exported XML file. Here's a simple example:

```
<?xml version="1.0"?>
<templateItems template="/Templates/planet.dwt"
codeOutsideHTMLIsLocked="false">
   <item name="diameter"><![CDATA[
      <P> 7926 miles
      ]]></item>
   <item name="moons"><![CDATA[
      <P> 1
      ]]></item>
   <item name="planetImage"><![CDATA[<IMG
SRC="assets/Images/earth.gif" ALIGN="TOP" WIDTH="72" HEIGHT="72"
VSPACE="0" HSPACE="0" ALT="Earth Photo" BORDER="0">]]></item>
   <item name="doctitle"><![CDATA[
<TITLE>Earth</TITLE>
]]></item>
   <item name="orbital_period"><![CDATA[
      <P> 365 days, 6 hours, 9 minutes, 13 seconds
      ]]></item>
</templateItems>
```

Link to Template

Note placement in file

The first thing to notice is the tag identifying the template from which the document was derived. Another important aspect is the order—or rather the lack of order—of the item entries. In the original file, the editable regions

appeared in this sequence: `doctitle`, `planetImage`, `diameter`, `orbital_period`, and `moons`. In the exported data file, they are written in this order: `diameter`, `moons`, `planetImage`, `doctitle`, and `orbital_period`. This is one of the major advantages of an XML file over a less structured layout of information—the order the data is written is irrelevant to the order of its final presentation.

Dreamweaver imports XML files written in both standard and editable region-based syntax. Either of the following two formats is supported:

```
<templateItems template="/Templates/planet.dwt"
codeOutsideHTMLIsLocked="false">...</templateItems>

<planet template="/Templates/planet.dwt">...</planet>
```

When the File, Import, XML into Template command is given and a properly formatted XML file is selected, the data and the template instance are merged and a new document is created immediately.

Automated Import to Template

Again, we're faced with a powerful command that does exactly what we want—except that it does it with only one document at a time. What's clearly needed here is a parallel to the Export XML extension—one that permits the same choices in scope (current document, open documents, selected files in site panel, or entire site) and stores all the HTML documents generated from importing XML files in a single folder. Constructing this extension takes even less time because much of the work (looping through the selected files, for example) has already been done once before. A key point to take away from this section—aside from how to build the function—is how you can leverage work in one extension to make another.

NOTE: The approach described in this section is not the only way to combine data and templates to create new pages. To see how you can generate HTML pages directly from a data source, look at Chapter 5, "Automating Static Page Production from a Data Source."

Let's start by defining how we expect the new extension, Import XML, to work:

NOTE: To View the completed code, see Listing 4-4 at the end of this section.

1. From the Extension dialog box, the user selects which XML files should be included in the operation.
2. The user also selects a folder to store the generated pages.
3. When the command executes, the program gathers the URL of each XML file.

4. The URL is passed to a function that first creates a blank document to hold the template instance and then imports the XML data.

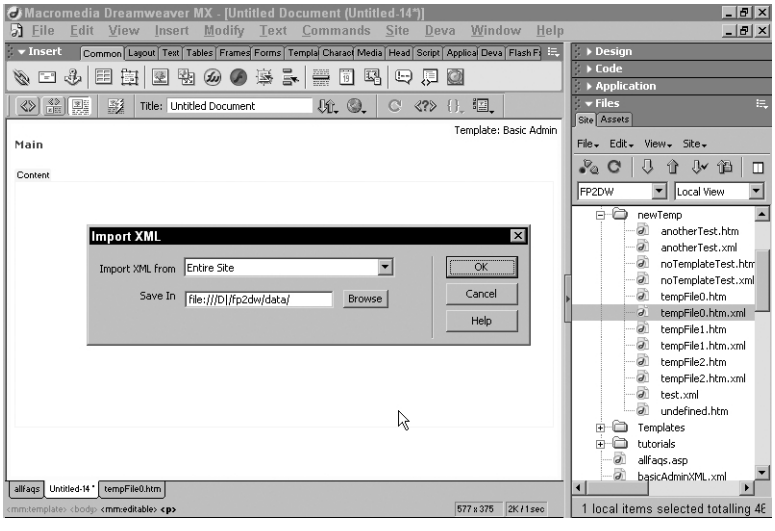
5. The new file is saved and closed—and the next XML file, if any, is processed.

The process and the Import XML extension are pretty straightforward. Best of all, we have a starting point to jump off from: the Export XML extension. In a situation like this where so much is duplicated from one extension to the next, I typically open the original extension and do a File, Save As to create a new extension. I save the new file in the appropriate Configuration folder; in this case, that would be Commands.

Next, I change the title of the extension. The title is displayed in the title bar of the Extension dialog box. Changing the title is a small thing, but I often forget to do it if I don't do it right away. Now we can begin to seriously modify the base extension, starting with the user interface. In this case, we'll take away an element found in Export XML that is not needed in Import XML: the Template drop-down list. For this version of Import XML, it is assumed that all the XML files declared their associated template. Why? The primary reason is that the key API function we will be using, `dw.importXMLIntoTemplate()`, requires it. It also greatly simplifies our coding.

With the Template drop-down list and its label deleted, the Import XML extension (see Figure 4.4) is ready for user input.

TIP: Remember that Dreamweaver MX now supports multiuser configurations. If you are working with a multiuser-compatible OS (such as Windows 2000 or Mac OS X), the custom extensions need to be saved in the appropriate user/Macromedia/Dreamweaver MX/Configuration folder. Extensions for single-user systems are stored in the Programs/Macromedia/Dreamweaver MX/Configuration folder (Windows) or in the Applications:Macromedia:Dreamweaver MX:Configuration folder (Macintosh).



4.4

Why rebuild when you can duplicate? The Import XML extension is a duplicate of Export XML with one user interface element removed.

I need to make only one small change to code within the `doCommand()` function of Import XML. In the Export XML extension, the `Wildcards` variable was set to allow almost every type of Web document extension: .htm, .html, .shtm, .shtml, .asp, .cfm, .cfml, .php, and .php3. For the Import XML extension, the selection needs to be limited to just one file type: .xml.

As before, the main function is separated from the code that takes the user's selection of which files to be processed. Here, that function is named `importXML()`, whereas in the original extension, it was named `exportXML()`. Because the user selection code calls the function for each choice of scope potentially made, the easiest way to modify the code is to do a find and replace.

The first line in the `importXML()` function serves to create a new, blank document:

```
var theTempDOM = dw.createDocument();
```

If this is not done, the template instance is loaded onto the current document.

The next bit of code represents one of the real pitfalls of extension programming: incorrect documentation. According to the *Extending Dreamweaver* manual, the `importXMLIntoTemplate()` function takes a file URL as its only argument. Unfortunately, that's wrong. Although it does take a URL pointing to an XML file, the string should be formatted as a file path, like this:

```
D:\fp2dw\newTemp\anotherTest.xml
```

instead of a file URL, like this:

```
file:///D|/fp2dw/newTemp/anotherTest.xml
```

Dreamweaver provides a function in the MMNotes API collection that converts a file URL to a file path. We can use that without including any other JavaScript file:

```
var theNewURL = MMNotes.localURLToFilePath(theURL);
```

With our new URL created, we're ready to perform the key operation of importing the XML into a template:

```
dw.importXMLIntoTemplate(theNewURL);
```

Now we create a new document name based on the XML filename and save the file. The new name substitutes an .htm extension for the file's original.xml one and incorporates the user-selected path to a folder:

```
docName = findObject("folder_text").value +
theURL.substring(theURL.lastIndexOf("/") + 1, theURL.length - 4) +
".htm";
res = dw.saveDocument(theTempDOM, docName);
```

Finally, if the save operation was successful, the document is closed and control passes back to the `doCommand()` function to get another URL for processing, if necessary.

```
if (res) {
   dw.closeDocument(theTempDOM);
}
return;
```

The most common error to watch for when using an extension like this is malformed XML. If you encounter problems, check the XML file by choosing Validate Current Document as XML from the Validation panel.

Listing 4-4 **Import XML** (04_importXML.mxp)

```
<html>
<head>
<title>Import XML</title>

<script language="javascript"
src="../Shared/Beyond/Scripts/Beyond_Site.js"></script>
<script language="JavaScript" src="../Shared/MM/Scripts/CMN/UI.js"
type="text/JavaScript"></script>
<script language='javascript'>
```

continues

```
//init globals ************************************************
var theSelect = findObject('fileList');
var theWildText = findObject('wildText');
var theXML = "";


//******************** Primary Functions *************************

function commandButtons(){
    return new Array( 'OK', 'doCommand()', 'Cancel',
    'window.close()', 'Help', 'getHelp(theHelpFile)')
}

function importXML(theURL) {
   var theTempDOM = dw.createDocument();
   var theNewURL = MMNotes.localURLToFilePath(theURL);
   dw.importXMLIntoTemplate(theNewURL);
   docName = findObject("folder_text").value +
   theURL.substring(theURL.lastIndexOf("/") + 1, theURL.length -
   4) + ".htm";
   res = dw.saveDocument(theTempDOM, docName);
   if (res) {
      dw.closeDocument(theTempDOM);
   }
   return;
}


function doCommand() {
   var selectArray = new Array("currentDoc","openedDocs",
   "siteSelected","wholeSite");
   var theRes = 1
   var theWildCards = ".xml"

   for (var i=0; i<theSelect.options.length; i++){
      if (theSelect.options[i].selected){
         whichFiles = selectArray[i];
```

```
    }
}
switch (whichFiles){

    case "currentDoc":
        urlArray = getCurrentDoc();
        if(urlArray){
            importXML(urlArray);
        }
        break;

    case "openedDocs":
        agree = confirm("This command cannot be undone.
        Proceed?");
        //If it's ok, go
        if (agree){
            openFilesArray = new Array();
            //Get the currently opened files
            openFilesArray = getOpenedDocs();
            //Filter them to get just the ones matching extensions
            urlArray = filterFiles(theWildCards,openFilesArray);
            for (var i=0; i<openFilesArray.length; i++){
                importXML(urlArray[i]);
            }
        } else {
            return;
        }
        break;

      case "siteSelected":
        var siteFocus,agree;
        siteSelectedArray = new Array();
        writeSiteSelectedArray = new Array();
        siteFocus = site.getFocus();
        if(siteFocus == "local" ¦¦ "site map"){
            //Ask the user
```

continues

```
            agree = confirm("This command cannot be undone.
            Proceed?");
            //If it's ok, go
            if (agree){
               //Get the urls of the files selected inside the
               //site window
                siteSelectedArray = site.getSelection();
               if (siteSelectedArray.length == 0 ||
               siteSelectedArray[0].indexOf(".") == -1) {
                  alert("No files in Site window selected.\nChoose
                  another Generate From option.")
                  return;
               }
               //Filter them to get just the matching extensions
               urlArray = filterFiles(theWildCards,
               siteSelectedArray);
               for (var i=0; i<urlArray.length; i++){
                  importXML(urlArray[i]);
               }
            } else {
               return;
            }
         }
         else{
            alert("This command can affect only local files");
         }
         break;

      case "wholeSite":
         var agree;
         wholeSiteArray = new Array();
         writeFilesArray = new Array();
         //Ask the user
         agree = confirm("This command cannot be undone.
         Proceed?");
         //If it's ok, go
         if (agree){
```

```
            //Get all the urls of files in the site with matching
            //extensions
            wholeSiteArray = getWholeSite();
            //Filter them to get just the matching extensions
            urlArray = filterFiles(theWildCards,wholeSiteArray);
            for (var i=0; i<urlArray.length; i++){
               importXML(urlArray[i]);
            }
         } else {
            return;
         }
         break;
   }
   window.close();
   return;
}

function findFolder() {
   findObject("folder_text").value = dw.browseForFolderURL();
}

function findFile() {
   findObject("file_text").value = dw.browseForFileURL("select");
}

function getTemplateList() {
   //returns a list of templates in site
   var theTemplateDir = dw.getSiteRoot() + "Templates/";
   var theTemplates = new Array();
   theTemplates = DWfile.listFolder(theTemplateDir + "*.dwt",
   "files");
   if (theTemplates){
      loadSelectList(theTemplateList,theTemplates);
   }
}
```

continues

```
function initUI() {
}

</script>
</head>

<body onLoad="initUI()">
<form name="theForm">
  <table border="0">
    <tr>
      <td nowrap> <div align="right">Import XML from</div></td>
      <td nowrap> <select name="fileList" style="width:220px">
           <option selected>Current Document</option>
           <option>All Open Documents</option>
           <option>Selected Files in Site Window </option>
           <option>Entire Site</option>
        </select> </td>
    </tr>
    <tr>
      <td><div align="right">Save In</div></td>
      <td><input name="folder_text" type="text" id="folder_text"
      style="width:155">
        <input type="button" name="Button" value="Browse"
        onClick="findFolder()"></td>
    </tr>
  </table>
</form></body>
</html>
```