

SYBEX Bonus Chapter

Developing Killer Web Apps with Dreamweaver MX[®] and C#[™]

Chuck White

Bonus Chapter 2: Working with XML

Copyright © 2004 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

ISBN: 0-7821-4254-0

SYBEX and the SYBEX logo are either registered trademarks or trademarks of SYBEX Inc. in the USA and other countries.

TRADEMARKS: Sybex has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer. Copyrights and trademarks of all products and services listed or described herein are property of their respective owners and companies. All rules and laws pertaining to said copyrights and trademarks are inferred.


This document may contain images, text, trademarks, logos, and/or other material owned by third parties. All rights reserved. Such material may not be copied, distributed, transmitted, or stored without the express, prior, written consent of the owner.

The author and publisher have made their best efforts to prepare this book, and the content is based upon final release software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturers. The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Sybex Inc.
1151 Marina Village Parkway
Alameda, CA 94501
U.S.A.
Phone: 510-523-8233
www.sybex.com



Bonus Chapter 2: Working with XML

- The fastest primer ever on XML
 - DTDs and schemas for rules enforcement
 - Using the asp:XML web control
 - Calling XML from code
 - The fastest primer ever on XSLT
 - Writing XML and XSLT using the XML web control
 - Writing XML and XSLT programmatically
 - Using a DataGrid to display XML data
- 

Ah, the hype. But really—what will XML actually be used for on your ASP.NET site? I can't answer that for you, but it's a good bet there are some things you can think of with hardly any effort at all. For one, you could build an XML-based navigation system featuring server-side includes. Or you could build a content management system, or perhaps work with Web Services to exchange data with partners (as shown in Chapter 8, "Web Services and Dreamweaver MX").

In addition, many databases now handle XML much more efficiently than in the past. SQL Server 2000 provides support for both extracting and querying XML documents, as well as tools for mapping relational data into XML. Microsoft Access 2003 goes one step better and contains a built-in facility for mapping relational data to XML. Oracle and IBM's DB2 also offer substantial XML support.

XML, a data markup language that makes it very easy to transport data, is on the face of it a very easy language to learn. It is also steeped in massive numbers of vocabularies that can make your head spin. The most important among those vocabularies is *XSLT*, the *Extensible Stylesheet Language*, which can be used to convert or transform an XML document into another format such as HTML.

In this chapter, I'm going to introduce the basic concepts of XML and XSLT and show how .NET handles them. Then, you'll be able to easily build a navigation system or other XML-based project on your own.

The Fastest Primer Ever on XML

```
<Tutorial><best style="humorless">This Book</best></Tutorial>
```

Look familiar? Of course it does. Looks like HTML. Smells like HTML. It doesn't, however, *act* like HTML.

HTML affects the presentation and marks up the content of a page. In other words, the focus of HTML is on how the page contents look (actually, that was not the original intent of HTML, but that's a different story). HTML does things like **this**. And *this*.

XML, on the other hand, is more interested in a document's data. XML lets authors write their own tags to describe data. This makes it easy to create an unlimited number of indexing capabilities, which in turn means great support for database access. So if you can describe something in your mind (and, sadly, not all of us can), you can describe data in XML.

Say you're thinking of cataloging all the great pizza joints in Chicago. Here's how you'd go about it:

```
<PizzaJoints>Pizza Info</PizzaJoints>
```

And don't forget that closing tag! XML insists on it. Okay, but we want more info than just a simple list. We should divide up the pizza joints into categories, like Thin Crust and Thick Crust. So we could do this:

```
<PizzaJoints ThickCrustyOne="yes">Giordano's</PizzaJoints>
<PizzaJoints ThickCrustyOne="yes">Nancy's Stuffed Pizzas</PizzaJoints>
<PizzaJoints ThickCrustyOne="yes">Edwardo's</PizzaJoints>
```

Maybe we want to add another attribute, like whether I can order one of these pizzas over the Web and have it sent to me in San Francisco:

```
<PizzaJoints ThickCrustyOne="yes" viaTheWeb="yes">Lou Malnati's </PizzaJoints>
```

You can't get easier than that. Of course, there are other ways to describe the same thing. But one step at a time here.

You might be wondering by now, "Is that all there is?" Well, think about it and you'll realize it's really quite a lot. XML is almost like having your own DTD-making machine, with which you can develop your own set of tags to describe data in just about any clever way you can think of. If you think this is small stuff, just look at the working draft of the new SVG (Scalable Vector Graphics) standard that has been submitted to the W3C. If you're a graphics buff who fantasizes of the day vector drawings and animations for the Web can be authored by anyone with a text editor, listen up. You might want to consider jazzing up your fantasies, and you'll need to upgrade your fantasy because such a day is not too far off.

Of course, there's much more to be done with XML. Database pros are drooling over it because you can create massive repositories that operate across any computer platform. At the risk of making it all sound too simple, imagine mapping the fields of a customer database to elements, and then being able to share that information across a vast array of platforms. That, in effect, is what the wildly hyped Web Services do; plus, they add to this by describing function calls in XML wrappers and sending them across disparate systems.

A Look at XML Document Construction

On a more technical level, an XML document consists of a Unicode-based character sequence built by patterns that govern the creation of a logical data hierarchy. The hierarchy is expressed according to conventions established by the XML 1.0 Recommendation, which can be found at <http://www.w3.org/XML/>.

For our purposes, the highlights of the XML 1.0 conventions are as follows:

- All elements must have opening and closing tags. None of this stuff allowed:

```

```

XHTML, which is an XML version of HTML, would treat the `img` element like this:

```

```

- Attributes must always have quotes; no exceptions, ever. No more of this:

```
<td colspan=2>
```

- Attributes must have values. You can't do this:

```
<checkbox checked>
```

- There must be one top-level element, and all elements must follow a hierarchical, ordered sequence and nesting pattern. Every element, except the top-level (root) element, must be a direct child of another element. So this would be illegal XML:

```
<elementOne>
<elementTwo>
</elementOne>
</elementTwo>
```

Instead, each element must distinctly contain the other. One element can't end before the other ends. Think of how you stack a series of plastic food containers; they fit inside each other neatly. XML elements work the same way. So the badly structured XML just above looks like this when written correctly:

```
<elementOne>
<elementTwo>
</elementTwo>
</elementOne>
```

Or, for the element `elementTwo`, you can use the shortcut like this:

```
<elementOne>
<elementTwo />
</elementOne>
```

When an element doesn't have any content, you can close it within the same tag that opens, as shown above.

- In element and attribute content, the following predefined entities must be used instead of the actual symbols they represent:

`<` for `<`

`>` for `>`

`&` for `&`

`'` for `'`

`"` for `"`

In other words, don't try to use (for example) the `<in` element content like this:

```
<elementOne>
  <elementTwo><elements></elementTwo>
</elementOne>
```

The XML parser will think you're trying to create markup—specifically, an element named `elements`—and will yell at you for not closing the element before closing its parent element, which would be `elementTwo`.

Using DTDs and Schemas to Enforce Rules

XML is interesting as a data markup language, but things can get weird when you start commingling your data with that of other people and environments. A book `<page>` to you might be a database `<page>` to someone else and a beeper `<page>` to yet another person. To control this, there are two processes for developing rules for XML documents: DTDs and schemas.

DTDs and XML

One type of rules enforcement uses *DTDs*, or *Document Type Definitions*, which are similar to the set of rules that defined all versions of HTML up to and including HTML 4.0. In fact, HTML markup is indeed ruled by a set of DTDs and is based on XML's ancestor language, SGML (Standardized General Markup Language). HTML is an SGML vocabulary governed by DTDs. Enforcement of these DTDs has always been inconsistent (hence the browser wars of a few years ago), but they do exist.

You can view the HTML DTD here: <http://www.w3.org/TR/html4/sgml/dtd.html>, where you'll see a lot of weird syntactical stuff like this:

```
<!ELEMENT IMG - O EMPTY
<!ATTLIST IMG
  %attrs;
  src      %URI;      #REQUIRED
  alt      %Text;     #REQUIRED
  longdesc %URI;      #IMPLIED

  name     CDATA      #IMPLIED
  height   %Length;   #IMPLIED
  width    %Length;   #IMPLIED
  usemap   %URI;      #IMPLIED
  ismap    (ismap)    #IMPLIED
>
```

An XML DTD contains markup that looks very similar to the preceding code fragment, which defines the HTML `IMG` element. All those words you see with leading `%` characters are known as *parameter entities*, which is just shorthand for rules that were defined earlier so you

can reuse the definitions. The definition of the `IMG` element includes a list of attribute definitions (`ATTLIST`). The `name` attribute, for example, is defined as consisting of some character data (`CDATA`). The `src` attribute is defined as consisting of the rules that were defined for the `URI` parameter entity, so to find out what the rules for `src` really are, you have to backtrack and find out the rules for the `URI` parameter entity. These common rules get applied all over the place, which is why the creators of the spec put them into parameter entities in the first place. The DTD also defines whether or not the attributes are required (through the `REQUIRED` keyword) or optional (through the `IMPLIED` keyword).

You can create an XML DTD consisting of rules that define the purpose, content, and value constraints of your elements and attributes. When you create a DTD, you save the file with the `.dtd` extension. Then you refer to it in your XML if you want to validate your XML document against the DTD you created. For example, here's an XML document:

```
<?xml version="1.0"?>
<!DOCTYPE MYDOC SYSTEM "mydoc.dtd">
<MYDOC>
  <DOC>Some content</DOC>
  <DOC2>More Content</DOC2>
</MYDOC>
```

The DTD for this XML document might look like this:

```
<!ELEMENT MYDOC (DOC, DOC2)>
<!ELEMENT DOC (#PCDATA)>
<!ELEMENT DOC2 (#PCDATA)>
```

Here, the root element, `MYDOC`, is defined as consisting of two elements, `DOC` and `DOC2`. Then each of those elements is defined as consisting of parsed character data, which is raw text that doesn't include any markup.

There are many DTDs currently in existence. For example, `XHTML`, which is a flavor of `HTML` designed to follow the guidelines of XML vocabularies, consists of DTDs that rigidly enforce a set of XML-based rules. If you develop `XHTML` documents that aren't marked up properly, a browser will not display them. Luckily, Dreamweaver MX provides a mechanism for generating `XHTML`-compliant code. There's a check box at the lower-right of the New Document dialog box labeled, "Make document `XHTML` compliant." Check this option when you create a new document, and Dreamweaver generates the following DTD declaration:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

The DTD referenced here is a transitional DTD, which enjoys better browser support than the strict DTD, which in turn is similar to the strict DTD for `HTML 4`, except that it is designed for XML compatibility.

Dreamweaver also adds a namespace to the `html` element:

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

NOTE

All XHTML elements *must* be all lowercase.

Many other kinds of DTDs are available. One of the more widely used DTDs is the DocBook DTD for creating publications; it can be found at <http://www.docbook.org/>.

It's always a good idea to check for an existing DTD before creating your own. This is easy to do. Simply choose your subject and type it and **DTD** into Google. If you search for **Recipe DTD**, for example, you'll find DTDs written for recipes.

For an elementary and easy-to-follow guide to DTDs, visit this site:

<http://www.ibiblio.org/xml/slides/sd2000east/dtds/>

Schemas and XML

The other process you can use for rules involves *schemas*. Schemas are gaining momentum because they provide richer data-typing capabilities, and their use of namespaces lends them well to Web Services.

Understanding Namespaces

So what about these namespaces? They are mechanisms for uniquely identifying the meaning of an XML document's nodes. They work through a binding mechanism using the `xmlns="some uri"` declaration in your XML document. The URI can be any URI, but often they are URLs to existing domains because the authors feel confident that nobody else will use them. A namespace has no physical location, however, so even though you might see something like this:

```
<root xmlns=http://www.tumeric.net/namespaces/">
```

it doesn't mean there's anything at that location. It merely means you've declared that the root element belongs to that namespace. Often, that namespace will include a schema definition, which is similar to a DTD but uses XML syntax to define rules. You'll see an example of a schema in this chapter, in the section "Using a DataGrid to Display XML Data." For more information on namespaces, visit this site:

http://www.rpbouret.com/xml/NamespacesFAQ.htm#q1_1

Understanding XML Schema

DTDs were nice, particularly for document-oriented XML, but many people found them limiting. So the W3C came up with the *XML Schema* language, which uses an XML-based syntax for defining rules, has enhanced datatyping, and allows you to do things like build referential XML-based databases.

A schema is referenced through common namespaces. One of these is defined in the `targetNamespace` attribute of the XML Schema root element, and the other is named in the namespace declaration of the XML document you want to validate. For example, here is a code fragment from a schema that I used for creating a `Listings.xml` document:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema elementFormDefault="qualified" targetNamespace="Listings.xsd"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:od="urn:schemas-microsoft-
com:officedata">
<xsd:element name="dataroot">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="dbo_Listings" minOccurs="0" maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="generated" type="xsd:dateTime"/>
</xsd:complexType>
</xsd:element>
```

In this example, the first element defined is named `dataroot` (in bold). It is defined as a `complexType`, which means that it consists of a sequence of other elements. Then the sequence of elements contained in this `dataroot` element is defined. The `dataroot` element contains only one child element, and this is referenced by the `xsd:element` element's `ref` attribute. This element is actually defined later in the document. It's only referenced here as being part of the content of the `dataroot` element. Then an attribute named `generated` is defined for the root element. Note that its datatype (`dateTime`) is also defined. DTDs don't have this kind of rich datatyping (integers, `dateTime`, floats, and so forth), but XML Schema does. You'll see how this all fits together in the later section on DataGrids, which harness the power of XML Schema to build table layouts.

For a more comprehensive primer on XML Schema, visit this website:

<http://www.w3.org/TR/xmlschema-0/>

ASP.NET and XML

A misguided notion about ASP.NET that tricks many people happens when they look at ASP.NET markup and mistakenly refer to it as XML. The markup for ASP.NET looks a lot like XML, and all of the web controls certainly seem to follow the standard conventions for XML markup. Take a look at the following code snippet:

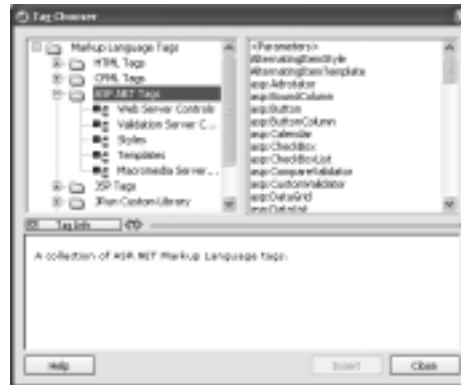
```
<asp:label ID="xLabel" runat="server" Text="This is a label"></asp:label>
```

This certainly looks like acceptable XML markup. The problem is, the following is also perfectly acceptable ASP.NET markup:

```
<asp:label ID="xLabel" runat=server Text="This is a label"></asp:label>
```


FIGURE 2.2:

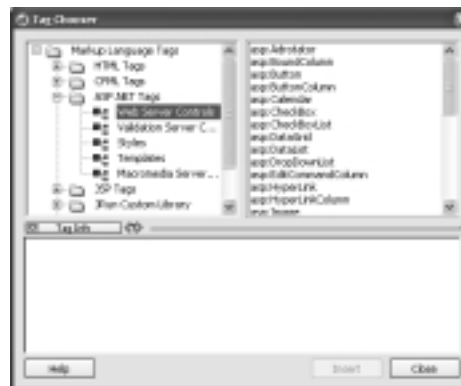
The ASPNET tags in the Tag Chooser dialog box



Highlight the Web Server Controls in the left hand panel of the Tag Chooser (Figure 2.3). Scroll down in the right-hand panel and choose `asp:Xml` (either double-click it or highlight it and click the Insert button). Notice that the Tag Chooser stays open, so it's easy to accidentally click the Insert button again.

FIGURE 2.3:

Highlighting the Web Server controls in the Tag Chooser



To fill out the Tag Editor dialog box that appears as a result of selecting the `asp:Xml` element, refer to Table 2.1, which describes the purpose of the Tag Editor fields for this element. For now, focus on the Document Source field and the Transform Source field. Also, give the control an ID value; it's a good habit to always set up a programmable reference to any control you add.

TABLE 2.1: XML Control/asp:Xml Element Options

Dreamweaver Dialog Box Text Field	Equivalent ASP.NET Element Attribute	Description	Values
ID	ID	Uniquely identifies the element.	A string value you choose.
Document	Document	Specifies the XML document using a System.Xml.XmlDocument object. The XmlDocument object loads a full DOM into memory, giving you programmatic access to every node in the XML tree. It also exposes a large number of public properties and methods.	The document object's name.
Document Content	DocumentContent	Specifies the XML document through a string; in other words, you can write the document out as a literal string such as <code><root><e1>test</e1></root></code> .	A string consisting of the XML you wish to include in the document.
Document Source	DocumentSource	Specifies the XML document using a filename.	A string naming the file.
Transform	Transform	Transforms the document using the XSLT code named in the string value of this attribute.	A string consisting of the XSLT you wish to use to transform the XML document.
Transform Source	TransformSource	Specifies an XSLT file that will be used to transform the XML document into another format, such as HTML.	A string naming the XSLT file.

In the Document Source field of the Tag Editor, enter a value of **Listings.xml**. In the Transform Source field, enter a value of **Listings.xsl**. Click OK, and click the Tag Chooser's Close button. Dreamweaver will insert code that looks like this:

```
<asp:Xml DocumentSource="Listings.xml" ID="xmlId" runat="server"
  TransformSource="Listings.xsl" />
```

The reason this is so simple is that there is very little programming to do. You don't even need to import ASP.NET XML namespaces, because the necessary architecture for handling the XML is built into the control. (In this case, I'm referring to the .NET kind of namespace, not the XML kind.)

You'll see how this renders an XML document in the section in this chapter on XSLT, which is a necessary component for rendering XML as HTML, unless you're using a data rendering component like a DataGrid.

Calling XML from Code

As your use of XML grows more sophisticated, you'll want to run your XML programmatically rather than from a control, because it will be easier to update your XML documents that way. The key to doing this is the use of two main classes, `XmlReader` and `XmlWriter`.

Using XML Readers

Although the complexities and options of XML readers could take up an entire book, for our purposes here we'll give you at least a general understanding of the most important concepts of the `XmlReader` class. It contains methods that pull data from XML sources. Using this class you can pull only the nodes you want, which saves you a lot in terms of memory because you don't have to load an entire XML tree right off the bat.

Because the `XmlReader` class is an abstract class, you actually won't refer to it often. (Abstract classes actually just define a way to create other classes that can be derived from the abstract class's various properties and methods.) .NET has three such defined classes built in, all of which you can use directly in your code.

The `XmlTextReader` class This is the fastest way to read an XML file. This class checks that the file *is* well formed XML but is *not* used to validate the XML.

The `XmlValidatingReader` class This class can be used to validate an XML document against a DTD, a schema with an `.xsd` extension, and XDR (the old Microsoft schema language).

The `XmlNodeReader` class This class exposes the XML Document Object Model (DOM) and can bring in a subtree rather than the entire document.

All of these classes have access to the public properties and methods of the `XmlReader` class, which can be viewed by examining the .NET documentation at

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfssystemxmlxmlreadermemberstopic.asp>

This URL will take you to the public properties and methods of the `XmlReader` class. Should Microsoft change the URL, you should be able to type **.NET `XmlReader`** into Google and find the definitions you need.

Let's take a look at an XML file, then, and see if we can learn something about it by using classes derived from the `XmlReader` class and some of the properties and methods available

through it. We'll study the XML file shown in Listing 2.1. Keep your eye on this Listings.xml file, because you'll be referring to it often in this chapter.

**Listing 2.1**

Listings.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<dataroot>
  <dbo_Listings>
    <MLSNumber>55782</MLSNumber>
    <Title>Gorgeous Fixer Upper</Title>
    <PropertyAddress>444 Contractors Lane</PropertyAddress>
    <City>San Francisco</City>
    <State>CA</State>
    <PostalCode>94444</PostalCode>
    <IDNumb>1</IDNumb>
    <DescriptionPlainText>A Fixer Upper in Upper Noe Valley</DescriptionPlainText>
    <Client_ID>2</Client_ID>
    <Price>600000</Price>
    <Neighborhood>Noe Valley</Neighborhood>
    <Status>Listed</Status>
  </dbo_Listings>
  <dbo_Listings>
    <MLSNumber>55178</MLSNumber>
    <Title>Beautiful Home</Title>
    <PropertyAddress>44 Beauty Lane</PropertyAddress>
    <City>San Francisco</City>
    <State>CA</State>
    <PostalCode>94444</PostalCode>
    <IDNumb>2</IDNumb>
    <DescriptionPlainText>A beautiful Noe Valley Home</DescriptionPlainText>
    <Client_ID>1</Client_ID>
    <Price>780000</Price>
    <Neighborhood>Noe Valley</Neighborhood>
    <Status>Sale Pending</Status>
  </dbo_Listings>
</dataroot>
```

The key to the whole process is simply creating an instance of one of the derived classes. Let's choose the XmlTextReader class:

```
XmlTextReader textReader = new XmlTextReader(Server.MapPath("Listings.xml"));
```

Our new object is called textReader, and it can now access every property and method available to the XmlTextReader class, which in turn inherits from the XmlReader class.

In typical .NET fashion, all the work from this point is really done by .NET. All we need to do is write the code that displays information that is already available to us. So I used a trusty tool from the old ASP days, the Response.Write method, to write out the values of the various XmlTextReader properties as they pertain to the Listings.xml document, as highlighted in bold in Listing 2.2.

**Listing 2.2****Reading the properties from the XmlTextReader class (Listings_xmlReader.aspx)**

```

<%@ Page Language="C#" ContentType="text/html" ResponseEncoding="iso-8859-1" %>
<%@ Import Namespace="System.Xml"%>

<html>
<head>
<script runat="server">
protected void Page_Load(Object Src, EventArgs E)
{
    // Create an instance of XmlTextReader and call Read method to read the file
    XmlTextReader textReader = new
    XmlTextReader(Server.MapPath("Listings.xml"));
    textReader.Read();
    Response.Write("<html>");
    Response.Write("<head><title>Using XmlReader</title>");
    Response.Write("</head><body>");
    // If the node has value
    while (textReader.Read() )
    {
        // Move to first element
        textReader.MoveToElement();

        Response.Write("XmlTextReader Properties
↳Test<br>");
        Response.Write("=====<br>");

        // Read this element's properties and
        //display them on the web page
        Response.Write("Name: " + textReader.Name + "<br>");
        Response.Write("Base URI: " + textReader.BaseURI
↳+ "<br>");
        Response.Write("Local Name: " +
↳textReader.LocalName
↳ + "<br>");
        Response.Write("Attribute Count: " +
↳textReader.AttributeCount.ToString() + "<br>");Response.Write("Depth: " +
↳textReader.Depth.ToString() + "<br>");
        Response.Write("Line Number: " +
↳textReader.LineNumber.ToString() + "<br>");
        Response.Write("Node Type: " +
↳textReader.NodeType.ToString() + "<br>");
        Response.Write("Attribute Count: " +
↳textReader.Value.ToString() + "<br>");
    }
    Response.Write("</body></html>"); }
</script>

```

If you've never been exposed to ASP, you may not be familiar with the `Response.Write()` method. This is a method from the `Response` object, which belongs to the `Page` class. It lets you write out text to the browser, including HTML markup. If you look at Figure 2.4, you can see part of the results from running Listing 2.2. From this book's page at www.sybex.com, you can download the HTML result file (which comes from doing a view source and saving it as an HTML file) to look at it more closely. The file is called `Listing0202_h.htm`.

FIGURE 2.4:

A browser revealing the `XmlTextReader` properties



Using XML Writers

Another base class for managing XML is the `XmlWriter` class. In the preceding section, you saw how several classes derived from the `XmlReader` class could be used to read XML trees. The same principles work with the abstract `XmlWriter` class. You won't be accessing the properties and methods of this class directly. Instead, you'll be accessing the properties and methods of classes that are derived from it, including any public properties and methods available through the class. Currently, the main class that falls into this category is `XmlTextWriter`. You could conceivably build your own class based on `XmlWriter`, and others may appear within the .NET framework, but for now `XmlTextWriter` is your main tool.

You can explore the public properties and methods for the `XmlWriter` class at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemxmlxmlwritermemberstopic.asp>

***XmlTextWriter* and Encoding Issues**

The `XmlTextWriter` class has only one potential flaw: Although it checks for “well-formedness” (XML-speak for syntactically correct XML) before you output an XML tree, there’s no check to make sure your encoding is okay. Therefore, if you’re outputting extended character sets (unusual characters, for example) into pages that can’t handle the encoding, you can run into trouble.

The best way around this is to make sure all your encodings are UTF-8. Since Dreamweaver defaults to iso-8859-1, you’ll need to check for consistency. If you aren’t using any unusual letters, you can just use the iso-8859-1 encoding everywhere. Whether you choose UTF-8 or iso-8859-1, problems may still occur when you use Windows encodings (which can happen when copying and pasting from Windows applications). That’s because Windows encodings have a few characters not within the Unicode spectrum.

For a more complete discussion of encodings as they relate to XML, I strongly recommend the rather esoteric but informative discussion at <http://skew.org/xml/tutorial/>. Don’t let yourself be put off by the high level of this encoding discussion. Skip past the stuff you don’t want to deal with and read enough to get a grasp of the basic issues. If you do, you’ll find yourself able to troubleshoot a large number of XML-related issues.

The main properties and methods you’ll be able to access using the `XmlTextReader` class can be seen at the following website:

```
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemxmlxmltextwriterclasstopic.asp
```

or by typing **`XmlTextReader`** .NET class into Google. We’ll see a couple of these methods and properties in action in the next section.

Displaying and Rendering XML

There are two ways of rendering XML, assuming you don’t want to simply deliver it in its raw form to the browser. You can use a `DataGrid` or one of the other data-based web controls (such as the `Repeater` control discussed in Chapter 3, “Working with Databases: An Introduction”); or you can use XSLT. As explained earlier, XSLT stands for Extensible Stylesheet Language, which is used for rendering XML documents as HTML and other text-based formats.

The advantage of rendering XML using a data component such as a `DataGrid` is that you can take advantage of the power of .NET to manipulate the data. The advantage of using XSLT is that it’s cross-platform, so you can use it among disparate systems.

We're going to take a look at using XSLT first, and then we'll see how the DataGrid can be used to present XML data in a table.

The Fastest Primer Ever on XSLT

You use XSLT to render XML documents into browser-readable HTML. You can also use it to render XML documents into other formats, including PDF files, SVG, and other text-based formats, but since this is the fastest primer ever on XSLT, we'll keep our focus on HTML.

XSLT can be very easy or very hard, depending on how fast you get used to it and how deeply you want to get into it. It's pretty easy to get a basic XSLT document out the door, but the language has a high level of complexity because it can be used as a full-fledged computer language.

If you've worked with other server-side languages such as ASP and, of course, ASP.NET, some of the principles of XSLT will be familiar to you.

In .NET, an XSLT processor evaluates your XSLT document and generates the results into a tree of nodes. We're going to take a brief look at how this is accomplished by working through one principal example and introducing a few of the major elements of XSLT. In your Dreamweaver development work, it's possible someone will give you an XSLT document and all you'll have to do is plug it in. However, if this isn't the case, the information in this chapter gives you an idea of how XSLT works.

Using XPath to Select Nodes

The key to understanding XSLT is getting to know another language that it uses, called XPath. XPath is one of those gnarly languages that we just have to get used to if we want to take full advantage of XSLT and certain kinds of DOM manipulation.

XPath can look really scary at first, but it's not too big a deal if you think of it in terms of an addressing scheme. Suppose you're going to the doctor for the first time and you need to know how to get there, so you ask for directions. Say the doctor's office is on Pierce Street. The directions to get there won't start off with "turn right on Pierce Street." Rather, they'll give you a starting point first. They'll say something like this: "Starting from Western Avenue, go north for about a mile and turn left on Randall Street. Look for Josh's Deli, then turn right onto Pierce Street. Our address is 111 Pierce St."

XPath works the same way, except you're addressing nodes in an XML document. A node can be any of these: an element, an attribute, a sequence one or more text characters (called a text node), a processing instruction, a comment, or a namespace. The entire document is also a node, and in XSLT this is called the root node (distinct from the root element, which is just one of what may be many element nodes).

You generally traverse an XML tree in XPath by using a series of / characters. Sometimes, if you just want to get to the next immediate child of the node from which you're referencing, or starting, you just name the node. Let's see how this works. Take another look at Listing 2.1, which shows the XML document named `Listings.xml`. Let's say we want to access all the data in the first `dbo_Listings` element. Consider the example of someone giving you directions. You'll be started off at your destination, just as you are in this XML document. We need to get to where we're going by following one node to another until we find our way to the destination (`dbo_Listings`).

The following gets us there no matter where we are when we start, because it automatically begins the journey at the root node:

```
/dataroot/dbo_Listings[1]
```

Whenever an XPath starts out with a leading /, this means that the journey begins at the root node of the document, no matter from where that XPath is referenced. You'll see later that in an XSLT document, you can find yourself in different positions of the XML tree at various times, so this root node beginning is an important consideration.

Usually, you'll then want to simply get to where you're going. For example, if you're already in the `dataroot` element but you want to get to the first `dbo_Listings` element's `MLSNumber` element, you would do this:

```
dbo_Listings[1]/MLSNumber
```

You don't need to name the `dataroot` element if you're already there.

Our example XML document doesn't include any attributes, but if, for example, the `MLSNumber` element had an attribute named `type` and you wanted to access its value, you would do this:

```
dbo_Listings[1]/MLSNumber/@type
```

Whenever you need to access an element's attribute, you precede the attribute name with the @ character so that the XPath or XSLT processor knows you're looking for an attribute. If you wanted to get to the second `dbo_Listings` element instead of the first, you would replace the `[1]` with `[2]`:

```
dbo_Listings[2]/MLSNumber/@type
```

You can use brackets like this to further refine your searches. The purpose of the number in brackets is not to simply name the node by the order it appears in sequence; rather, the brackets are a filtering device. The expressions within these brackets can get quite complex, so let's look at an easy one. Assume that we know a specific `MLSNumber`. We can filter out the `dbo_Listings` element we want by checking to see if it contains any `MLSNumber` elements equal to a specific MLS number:

```
dbo_Listings[MLSNumber = '55782']/MLSNumber
```

The expression in brackets is a way to more fully refine your search. You're saying, "Give me the `MLSNumber` element that is a child of the `dbo_Listings` element that contains a child `MLSNumber` whose value is equal to 55782." As we've warned you, the XPath language can get quite complex, and so can the expressions you create for filtering out nodes you wish to address.

The specification for XPath can be found here: <http://www.w3.org/TR/xpath.html>. As this book went to press, XPath was at Version 1.0, and Version 2.0 was still in development.

XSLT Template Instructions

XSLT is driven by a set of rules you create for a given set of nodes. What you're trying to say is that when a certain node from within an XML document is encountered, some specific thing should happen. Remember that we said XML is really just a set of rules for governing the way sequences of characters are output. This is important to remember when working with XSLT. A rule can get pretty fine-grained. You could create a rule that says for every line break encountered in some XML content, add a `
` element. Or a rule might be much more extensive and broad-based and say that every time a `dbo_Listings` element is encountered, a certain set of HTML elements should be processed.

The main way of achieving this arrangement is through the `xs1:template` element. When you define a set of rules within an `xs1:template` element, you tell the XSLT processor that the rules are available for use. Templates can be created for specific nodes.

The best way to demonstrate this is to examine Listing 2.1. Notice that there are two `dbo_Listings` elements. Can you see why the `dbo_Listings` element would be a prime candidate for a series of table rows? We can create a rule that outputs table rows like so:

```
<xs1:template match="dbo_Listings">
  <tr>
    <xs1:for-each select="*">
      <td valign="top" style="border-bottom: 1px blue solid; border-
right: 6px #cccccc solid">
        <xs1:value-of select="." />
      </td>
    </xs1:for-each>
  </tr>
</xs1:template>
```

We know we want `<tr>` elements for each instance of the `dbo_Listings` element, so we define the rule by starting with an opening and closing `tr` tag:

```
<tr>
</tr>
```

We could stop there, although we'd end up with no table cells. The next step could be to process that template rule. To do that, we can write an `xs1:apply-templates` element within

the root template of the XSLT document. Notice that we need the full path of the node we want to select—now the `apply-templates` statement will search for any templates with a match that ends with a `dbo_Listings` element:

```
<xsl:template match="/">
  <xsl:apply-templates select="dataroot/dbo_Listings"/>
</xsl:template>
```

A template with a `match="/"` attribute value pair is sort of the mother of all the templates and is processed automatically. So if you omit it from your stylesheet, the XSLT processor pretends it's there anyway. XSLT will then automatically generate text output for all the nodes in the document unless you do something to stop it. In this case, it's enough to simply add the `xsl:apply-templates` element. If there is no relevant data in the matching template, no data will be output. In our case, there is very little data in the one template that matches the node for this `select` statement, the `dbo_Listings` node. All we've done is write out a couple of `tr` elements. Each `dbo_Listings` node will be processed. In this case there are two, so the output would look like this:

```
<tr>
</tr>
<tr>
</tr>
```

We'll need to add to that content, but before we do, take a look at Table 2.2 to review some of the important template-processing elements.

TABLE 2.2: XSLT Template Instruction Elements

Element	Description
<code>xsl:template match="..." name = "...", mode="..."</code>	Defines the rules for a set of nodes. These rules wait until something processes them. The nodes get processed by either an <code>xsl:apply-templates</code> element or an <code>xsl:call-template</code> element. This element's <code>match</code> attribute is used to name the nodes to which your defined rules apply, and the <code>name</code> attribute is used by the <code>xsl:call-template</code> element to call a template by name. The <code>name</code> and <code>match</code> attributes cannot be used together; either one or the other must be used. The <code>mode</code> attribute can be used to specify a template when you define more than one set of rules for a matched set of nodes.

Continued on next page

TABLE 2.2 CONTINUED: XSLT Template Instruction Elements

Element	Description
<code>xsl:apply-templates select="..."</code>	Processes the nodes specified in the <code>select</code> attribute according to the rules defined by any matching templates.
<code>xsl:call-template name="..."</code>	Processes the template named in the <code>name</code> attribute.

XSLT Data Manipulation Elements

Previously I mentioned we'd need to add to our `dbo_Listings` template. But first, we need to know a little something about how to insert data into a result tree using XSLT. One way to do this is by using the `xsl:value-of` element, which will take all the text values of the node named in the `select` attribute and concatenate them into one long string before inserting the string into the result tree. And when I say *all* the text values of the node, I mean *all*. To demonstrate, let's temporarily comment out the `tr` element that we just created so the browser won't read malformed HTML:

```
<xsl:template match="dbo_Listings">
  <!--
  <tr>
  </tr>
  -->
  <xsl:value-of select="." />
</xsl:template>
```

The dot in the `select` attribute means that the context node should be selected. Now, take a look at Figure 2.5 to view the result of this. Oops, not quite what we want. But it's a start. Before we refine this further, review some of the data-manipulation elements that are available, as described in Table 2.3.

FIGURE 2.5:

Because we need to refine our query, the value of each and every text node in the `dbo_Listings` element appears in the results.

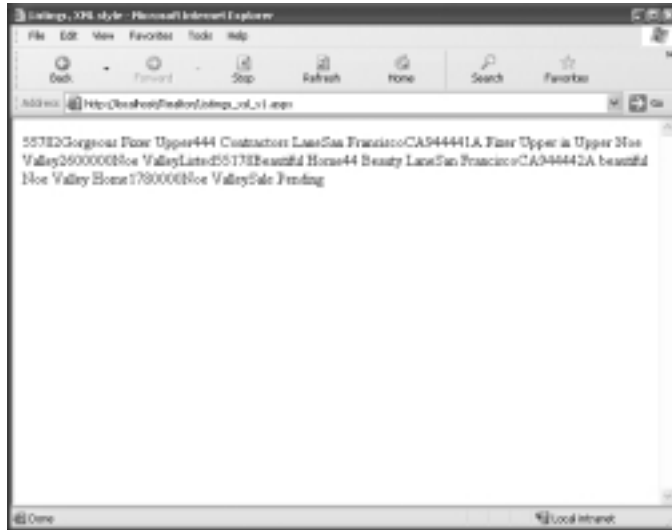


Table 2.3: XSLT Data Manipulation Elements

Element	Description
<code>xsl:value-of select="..."</code>	Returns the text value of a given node. Be careful, because if you return the value of a nested element, the text value of every element in the nested element will be returned as a concatenated value. Filter out exactly what you need by using an XPath statement in the <code>select</code> attribute.
<code>xsl:copy-of select="..."</code>	Returns the tree of the node named in the <code>select</code> attribute. If you were to do an <code>xsl:copy-of</code> on the <code>dataroot</code> element in Listing 2.1, it would return the entire document, minus the prolog (<code><?xml version="1.0" encoding="UTF-8"?></code>), because it copies the <code>dataroot</code> node, and all of its child nodes. When you select a node, you are also selecting that node's child nodes.
<code>xsl:sort select="...", order = "...", data-type="...", case-order = "..."</code>	Allows you to sort the ordering sequence of the result tree. Using the <code>select</code> attribute lets you specify which node within the context of the node the <code>xsl:sort</code> the sort should be made on; otherwise, the sort is made on the context node. The <code>order</code> attribute lets you choose between ascending and descending order, so you may also want to name the datatype by using the <code>data-type</code> attribute, which will generally be either the value <code>text</code> or the value <code>number</code> . The <code>case-order</code> attribute lets you choose whether the sort should be based on upper- or lowercase, with the possible values of <code>upper-first</code> and <code>lower-first</code> .

XSLT Control Flow Elements

We need to get rid of all that text shown in Figure 2.5. To do that, we must refine our selection process. So let's explore a bit further into the language.

Think about the problem for a moment. We've figured out how to generate a `tr` element for each `dbo_Listings` element, but now we're kind of stuck in that node. How do we then process each node within the `dbo_Listings` element? We could write a template for each node, or a universal template called an *identity transform*, by matching on all elements. But the easiest way to do it is to simply say, "For each child within the `dbo_Listings` element, I would like to generate a table cell and some appropriate content with each table cell."

Your next task is to review Table 2.4 and see if you can figure out which element will accomplish this generation of table cells and content (and remember to uncomment those `tr` tags).

TABLE 2.4: XSLT Control Flow Elements

Element	Description
<code>xs1:for-each select="..."</code>	This element contains instructions that are applied for each instance of a node selected by the <code>select</code> attribute. Note that the context node changes with this <code>select</code> statement. The context node becomes the node selected by this element, then reverts to its former self when the <code>xs1:for-each</code> element is closed. In addition to managing flow, you can use this element as a device for managing and changing context nodes.
<code>xs1:if test="..."</code>	If the expression in the <code>test</code> attribute returns true, then elements within the <code>xs1:if</code> element are processed, but only if the expression returns true.
<code><xs1:choose></code>	Processes the elements within the <code>xs1:when</code> statements (there can be more than one nested within an <code>xs1:choose</code> statement) when the <code>xs1:when</code> element's <code>test</code> attribute returns true. After the series of <code>xs1:when</code> statements are evaluated, you can supply a final option for processing elements with an <code>xs1:otherwise</code> element, which must come after any <code>xs1:when</code> elements.
<pre> <xs1:when test="..."> ... </xs1:when> <xs1:otherwise> ... </xs1:otherwise> </xs1:choose> </pre>	

If you chose the `xsl:for-each` element as the one to generate the table cells, you were dead on. So let's build a statement that says "for each child node of the `dbo_Listings` element, generate some table cells."

```
<xsl:for-each select="*">
  <td valign="top" style="border-bottom: 1px blue solid; border-
right: 6px #cccccc solid">
    <xsl:value-of select="." />
  </td>
</xsl:for-each>
```

While I was at it, I added some content for each table cell. As described in Table 2.4, the context node changes when you use a `for-each` statement to the selected node. Therefore, when the `xsl:value-of` element is used, it gives us the value of each child node. The `*` character is similar to a wildcard operator, in that it selects each child node of the context node. So the XSLT processor will iterate through the collection of the nodes that are children of the `dbo_Listings` element, and generate the output that exists within the `xsl:for-each` element.

Before we're finished, we need to wrap up the root template, so let's go back to that. We now have `tr` elements, and a series of `td` elements to go within those; but remember that we have to wrap these inside a table element in order for any legitimate HTML to be generated. The necessary changes are highlighted in bold:

```
<xsl:template match="/">
  <table cellpadding="5" border="0">
    <xsl:apply-templates select="dataroot/dbo_Listings" />
  </table>
</xsl:template>
```

Let's make one more change, and add some table cells. I'm cheating here and using an XPath function you haven't heard of (unless you know the language or were absurdly diligent about looking up more information on XPath. The `name()` function gives us the name of an element so that we can include it as part of table header cells. The changes are in bold:

```
<xsl:template match="/">
  <table cellpadding="5" border="0">
    <tr style="background-color:black; color:white">
      <xsl:for-each select="dataroot/dbo_Listings[1]/*">
        <th>
          <xsl:value-of select="name()" />
        </th>
      </xsl:for-each>
    </tr>
    <xsl:apply-templates select="dataroot/dbo_Listings" />
  </table>
</xsl:template>
```

XSLT Elements for Producing Markup

There are also a number of elements used for generating markup. You can generate elements using the `xsl:element` element, along with attributes:

```
<xsl:element name="body">
  <xsl:attribute name="bgcolor">#ffffff
</xsl:attribute>
<p>This is text</p>
</xsl:element>
```

This will result in the following result tree fragment:

```
<body>
  <p>This is text</p>
</body>
```

Notice that I mixed what are called *literal result elements*, as represented by the `p` element, with elements created using the `xsl:element` and `xsl:attribute` elements. This kind of mixing is perfectly okay.

Table 2.5 shows some of the elements that can be used for generating markup.

TABLE 2.5: XSLT Elements for Producing Markup

Element	Description
<code>xsl:element name="..."</code>	Generates elements into the result tree. The name attribute gives the name of the element that is generated, and any content within the element becomes the content of the new element.
<code>xsl:attribute name="..."</code>	Generates attributes into the result tree. The name attribute gives the name of the attribute that is generated, and any content within the element becomes the value of the new attribute.
<code>xsl:processing-instruction name="..."</code>	Generates processing instructions (PIs) into the result tree. The name attribute gives the name of the PI that is generated, and any content within the element becomes the value of the processing instruction.
<code>xsl:comment</code>	Generates comments into the result tree. Note: If you want comments that are ignored by the XSLT processor, use the <code><!-- --></code> tags that you would find in HTML comment markup.

Listing 2.3 shows what our final XSLT document looks like.

Listing 2.3 Transforming the Listings.xml document into some HTML output (Listing.xsl)

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" indent="yes" version="1.0"/>
  <xsl:template match="/">
    <table cellpadding="5" border="0">
      <tr style="background-color:black; color:white">
        <xsl:for-each select="dataroot/dbo_Listings[1]/*">
          <th>
            <xsl:value-of select="name()" />
          </th>
        </xsl:for-each>
      </tr>
      <xsl:apply-templates select="dataroot/dbo_Listings" />
    </table>
  </xsl:template>
  <xsl:template match="dbo_Listings">
    <tr>
      <xsl:for-each select="*">
        <td valign="top" style="border-bottom: 1px blue solid; border-
right: 6px #cccccc solid">
          <xsl:value-of select="." />
        </td>
      </xsl:for-each>
    </tr>
  </xsl:template>
</xsl:stylesheet>

```

Note the use of the `xsl:output` element. Use this element's `method` attribute to denote whether the output is HTML or XML. When you use the `method="html"` attribute value pair, the XSLT processor will generate proper HTML. This is important, because as you write your XSLT, you must write legal XML. So you can't write `
` elements into your XSLT document without closing them. You have to write things like `
` instead. Luckily, the XSLT processor knows how to convert that into HTML.

I didn't cover every XSLT element that is available. To really get a handle on the full power of XSLT will take some practice on your part. You'll find many resources all over the Internet if you wish to learn more about XSLT.

For a comprehensive review of XSLT, you can read *Mastering XSLT* (Sybex, 2000) or go to the specification itself:

<http://www.w3.org/TR/xslt>

You can also visit Microsoft's XSLT Developer's Guide site at

http://msdn.microsoft.com/library/en-us/xmlsdk/htm/xsl_intro_7yw5.asp

You can also get all your XSLT questions answered from one of the best developer communities I've seen, the XSL-List at

<http://www.mulberrytech.com/xsl/xsl-list>

Writing XML and XSLT Using the XML Web Control

As you saw earlier, you can use the XML web control to handle your XML transformations. I didn't then go into the process of working with XSLT, but we're going to get into that soon. First take a look at Listing 2.4 to see how an XML document is called from the ASP.NET page, and then the XSLT document itself. This is the easiest, fastest way to get an XML file to display using XSLT.

Of course, the hard work is in coding the XSLT. If you are part of a large staff, you'll likely have an XSLT expert on board, but if not, you'll need to learn the language.



Listing 2.4 Using an XML web control to display XML as HTML (Listings2_xml.aspx)

```
<%@ Page Language="C#" ContentType="text/html" ResponseEncoding="UTF-8" %>
<html>
<head>
<title>Listings, XML style</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
<body>
<asp:Xml DocumentSource="Listings.xml" ID="xmlId" runat="server"
TransformSource="Listings.xsl" />
</body>
</html>
```

You've seen this code already, in the earlier section "Using the *asp:XML* Web Control." Here, you can see its relationship to the rest of the document that uses it. Note that I changed the `ResponseEncoding` attribute in the Page directive to UTF-8 in order to maintain coding consistency.

Writing XML and XSLT Programmatically

You can also access and render your XML document using the `XmlTextWriter` class, which is derived from the `XmlWriter` class we looked at earlier. To access the XML document, I'm going to load it through a class you haven't yet seen, the `XPathDocument` class, which is similar to an XML reader but is not derived from that class in any way. The `XPathDocument` class comes from the `XPath` namespace, so you need to import that namespace into your code. The `XPathDocument` class is specially designed for fast access and manipulation through `XPath`, so it's a good choice for using with XSLT, because XSLT uses `XPath` to find nodes.

The other class we'll need also requires an imported namespace. The `XslTransform` class's role is to load the XSLT and use it to transform the XML into HTML output.

Listing 2.5 loads an XML document and transforms it using XSLT. Note that you need to import a few additional namespaces, because the `XPathDocument` class belongs in the `XPath` Namespace; and we're using a string writer class, which requires the `Text` namespace.



Listing 2.5 Accessing and rendering an XML document programmatically (Listings_xml.aspx)

```
<%@ Page Language="C#" ContentType="text/html" ResponseEncoding="iso-8859-1" %>
<%@ Import Namespace="System.Xml"%>
<%@ Import Namespace="System.Xml.XPath"%>
<%@ Import Namespace="System.Xml.Xsl"%>
<%@ Import Namespace="System.Text"%>

<html>
<head>
<script runat="server">
protected void Page_Load(Object Src, EventArgs E)
{
    XPathDocument doc = new
XPathDocument(Server.MapPath("Listings.xml"));
    XslTransform trans = new XslTransform();
    trans.Load(Server.MapPath("Listings.xsl"));

    StringWriter sw2 = new StringWriter();
    XmlTextWriter xmlWriter = new XmlTextWriter(sw2); //Write to
StringWriter

    xmlWriter.Formatting = Formatting.Indented;
    xmlWriter.Indentation = 4;
    trans.Transform(doc,null,xmlWriter);
    this.txtXmlTextReader.Text = sw2.ToString();
    sw2.Close();
    xmlWriter.Close();

}
</script>
<title>Untitled Document</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
<body>
<asp:label ID="txtXmlTextReader" runat="server"></asp:label>
</body>
</html>
```

This is one case where Dreamweaver doesn't help much. There is no direct support for these XML-related classes in the `DreamweaverCtrls.dll`. But that's okay, we're tough. We can handle this. This is not hard—because again, .NET really does all the hard work. All we need to do is know some of the syntax necessary to harness its power.

In this case, we create a new instance of the `XPathDocument` class, just as we created a new instance of an `XmlTextReader` class earlier in the chapter:

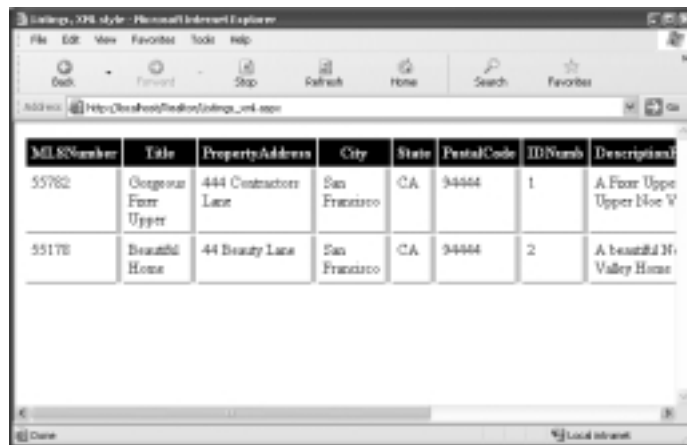
```
XPathDocument doc = new XPathDocument(Server.MapPath("Listings.xml"));
```

Then we use the `XslTransform` class to load an XSLT document, called `Listings.xsl`.

Even though the results are the same as using the XML web control (running either one should give you a web page similar to that in Figure 2.6), when you display XML programmatically you can include all your HTML tags in the XSLT document. Then you can just omit all the HTML elements in your ASPX page. This makes your XSLT more portable and decreases your reliance on ASP.NET or any other proprietary scripting language. However, in order to do that you have to rethink the code a little. That's because in our example we output the transformed XML file into a label, and in order to do that we had to plunk that label into some HTML. So we'll need to stream an entire HTML output instead of just a portion. You'll see how to do that in the last section of the chapter, coming up.

FIGURE 2.6:

`Listings.xml` looks the same in a browser whether we use the XML web control or call XML-related classes programmatically.



MLSNumber	Title	PropertyAddress	City	State	PostalCode	IDNumber	Description
55762	Gorgeous Four Upper	444 Contractors Lane	San Francisco	CA	94444	1	A Four Upper Upper Nice V
55176	Beautiful Home	44 Beauty Lane	San Francisco	CA	94444	2	A beautiful H Valley Home

Again, you'll have to plan ahead, so the principles outlined in Bonus Chapter 1, "Developing a Workflow," are once again thrust before us. What is the shape of our XSLT? Do we need to output an entire HTML document, including all the various meta information? Or do we need only an HTML fragment?

Using a DataGrid to Display XML Data

This section focuses on the `DataGrid` control, although any data reader control can be used to read an XML file. The `DataGrid` control is probably the best choice because it's so robust.

The DataGrid control does not use any of the XML reader or writer classes to read data. Instead, the DataGrid is actually a part of ADO.NET and relies on a method from the DataSet class to read the data. This method is called ReadXml, which can read XML files from a number of different sources, including files and XML text streams.

If you look at an XML file, you'll see that it isn't obvious on first glance how ASP.NET might decide to arrange the data in your document. You can use a schema document to help ASP.NET configure the way the columns should appear. (Schemas are discussed earlier in this chapter.) Listing 2.6 shows a schema generated by Microsoft Office 2003 using an Access database.



Listing 2.6

Listings.xsd generated from Microsoft Access database

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema elementFormDefault="qualified" targetNamespace="Listings.xsd"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:od="urn:schemas-microsoft-
com:officedata">
<xsd:element name="dataroot">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="dbo_Listings" minOccurs="0" maxOccurs="unbounded" />
</xsd:sequence>
<xsd:attribute name="generated" type="xsd:dateTime" />
</xsd:complexType>
</xsd:element>
<xsd:element name="dbo_Listings">
<xsd:annotation>
<xsd:appinfo>
<od:index index-name="pk_MLSNumber" index-key="MLSNumber" primary="yes"
unique="yes" clustered="no" />
</xsd:appinfo>
</xsd:annotation>
<xsd:complexType>
<xsd:sequence>
<xsd:element name="MLSNumber" minOccurs="1" od:jetType="text"
od:sqlType="nvarchar" od:nullable="yes">
<xsd:simpleType>
<xsd:restriction base="xsd:string">
<xsd:maxLength value="7" />
</xsd:restriction>
</xsd:simpleType>
</xsd:element>
<xsd:element name="Title" minOccurs="0" od:jetType="text"
od:sqlType="nvarchar">
<xsd:simpleType>
<xsd:restriction base="xsd:string">
<xsd:maxLength value="50" />
</xsd:restriction>
</xsd:simpleType>
```

```
</xsd:element>
<xsd:element name="PropertyAddress" minOccurs="0" od:jetType="text"
od:sqlSType="nvarchar">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:maxLength value="50"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
<xsd:element name="PropertyAddress2" minOccurs="0" od:jetType="text"
od:sqlSType="nvarchar">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:maxLength value="50"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
<xsd:element name="City" minOccurs="0" od:jetType="text" od:sqlSType="nvarchar">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:maxLength value="50"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
<xsd:element name="State" minOccurs="0" od:jetType="text"
od:sqlSType="nvarchar">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:maxLength value="50"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
<xsd:element name="PostalCode" minOccurs="0" od:jetType="text"
od:sqlSType="nvarchar">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:maxLength value="50"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
<xsd:element name="IDNum" minOccurs="1" od:jetType="autonumber"
od:sqlSType="int" od:autoUnique="yes" od:nonNullable="yes" type="xsd:int"/>
<xsd:element name="DescriptionPlainText" minOccurs="0" od:jetType="memo"
od:sqlSType="ntext">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:maxLength value="536870910"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
<xsd:element name="DescriptionHTML" minOccurs="0" od:jetType="memo"
od:sqlSType="ntext">
```

```

<xsd:simpleType>
<xsd:restriction base="xsd:string">
<xsd:maxLength value="536870910"/>
</xsd:restriction>
</xsd:simpleType>
</xsd:element>
<xsd:element name="DescriptionXML" minOccurs="0" od:jetType="memo"
od:sqlSType="ntext">
<xsd:simpleType>
<xsd:restriction base="xsd:string">
<xsd:maxLength value="536870910"/>
</xsd:restriction>
</xsd:simpleType>
</xsd:element>
<xsd:element name="Client_ID" minOccurs="0" od:jetType="longinteger"
od:sqlSType="int" type="xsd:int"/>
<xsd:element name="Price" minOccurs="0" od:jetType="longinteger"
od:sqlSType="int" type="xsd:int"/>
<xsd:element name="Neighborhood" minOccurs="0" od:jetType="text"
od:sqlSType="nvarchar">
<xsd:simpleType>
<xsd:restriction base="xsd:string">
<xsd:maxLength value="50"/>
</xsd:restriction>
</xsd:simpleType>
</xsd:element>
<xsd:element name="Status" minOccurs="0" od:jetType="text"
od:sqlSType="nvarchar">
<xsd:simpleType>
<xsd:restriction base="xsd:string">
<xsd:maxLength value="50"/>
</xsd:restriction>
</xsd:simpleType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>

```

The `targetNamespace` attribute at the top of the file refers to the URI of the schema. When it matches the `xmlns` in an XML document, the document can be validated against that schema.

ASP.NET can read a schema and infer from its rules the location of particular elements in a table. Building such a schema can be quite a challenge, so it's helpful to have a tool such as Microsoft Office to extract XML out of a database and write the schema for you. Keep in mind that Access versions prior to Access 2002 will not do this for you. (Office 2003 was still in beta as this book went to press.)

WARNING In order for ASP.NET to work correctly with a schema that uses a `targetNamespace` attribute, you must add the `elementFormDefault="qualified"` attribute/value pair to the schema's root element. Access will not add this for you.

If you don't have a schema, then the `DataSet` object will spit out the XML into a table according to the following criteria:

- Any elements with attributes will become tables.
- Any elements that contain other elements will become tables.
- Two or more elements that have the same name will become a table.
- All direct child elements of the root node become tables.
- Everything else becomes a column.

Listing 2.7 shows how we can load an XML document into a `DataGrid`. Note that I changed the XML document being accessed from `Listings.xml` to `Listings_2.xml`, which can be found with this book's other listings at the `www.sybex.com` site. The only difference between the two documents is that `Listings_2.xml` has a namespace binding to a schema named `Listings.xsd`, which was shown in Listing 2.6.

The results of running Listing 2.7 can be seen in Figure 2.7.



Listing 2.7 Loading an XML Document into a `DataGrid` (`DataGrid_xml_2.aspx`)

```
<%@ Page Language="C#" ContentType="text/html" ResponseEncoding="UTF-8" %>
<%@ Import Namespace="System.Data" %>
<html>
<head>
<script runat="server">
protected void Page_Load(Object Src, EventArgs E)
{
    DataSet oDS = new DataSet();
    oDS.ReadXml(Server.MapPath("Listings_2.xml"));
    dgDataGrid.DataSource = oDS;
    dgDataGrid.DataBind();
}
</script>
<title>DataGrids and XML</title>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
</head>
<body>

    <asp:DataGrid
        AutoGenerateColumns="true"
        AllowPaging="false"
```

```

ID="dgDataGrid"
PageSize="10"
runat="server"
ShowHeader="true"
Font-Name="Verdana"
Font-Size="11px"
HorizontalAlign="Center"
ItemStyle-BackColor="#FFFFCC"
AlternatingItemStyle-BackColor="#EEEEEE">
  <HeaderStyle BackColor="#333333" HorizontalAlign="Center"
    ForeColor="White" Font-Bold="True" />

</asp:DataGrid>

</body>
</html>

```

FIGURE 2.7:

An XML-based
DataGrid as seen
in the browser

#	Name	Title	Property Address	City	State	Postal Code	# of Rooms	Description	Floor Total
55782	Gorgeous Rustic Upper	444 Contractors Lake	San Francisco	CA	94444	1	2	A Finer Upper in Upper New Valley	2
55178	Beautiful New Valley Home	44 Beauty Lane	San Francisco	CA	94444	2	1	A beautiful New Valley Home	1

You can also control the way the DataGrid displays the data so that it doesn't load all the data by picking and choosing which columns you wish to display. This next code fragment shows the same DataGrid from Listing 2.7, but we've changed the `AutoGenerateColumns` attribute value to `false` (in bold) to prevent the DataGrid from grabbing every single field it finds based on the schema we created:

```

<asp:DataGrid
AutoGenerateColumns="false"
AllowPaging="false"
ID="dgDataGrid"

```

```

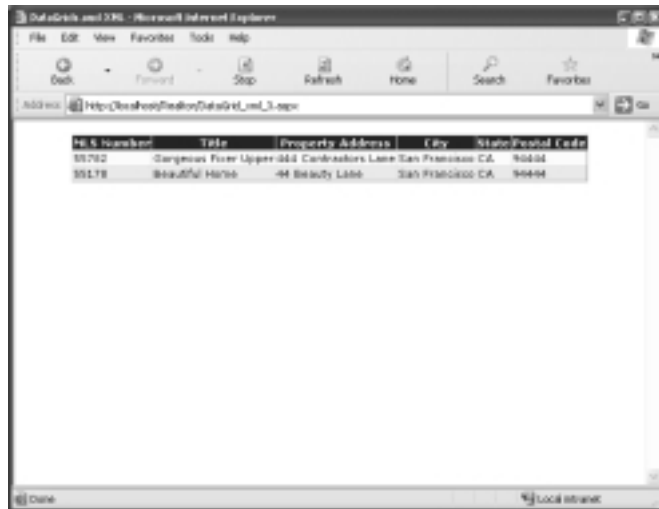
    PageSize="10"
    runat="server"
    ShowHeader="true"
    Font-Name="Verdana"
    Font-Size="11px"
    HorizontalAlign="Center"
    ItemStyle-BackColor="#FFFFCC"
    AlternatingItemStyle-BackColor="#EEEEEE">
    <HeaderStyle BackColor="#333333" HorizontalAlign="Center"
      ForeColor="White" Font-Bold="True" />
    <Columns>
      <asp:BoundColumn HeaderText="MLS Number" DataField="MLSNumber" />
      <asp:BoundColumn HeaderText="Title" DataField="Title" />
      <asp:BoundColumn HeaderText="Property Address"
DataField="PropertyAddress" />
      <asp:BoundColumn HeaderText="City" DataField="City" />
      <asp:BoundColumn HeaderText="State" DataField="State" />
      <asp:BoundColumn HeaderText="Postal Code" DataField="PostalCode" />
    </Columns>
</asp:DataGrid>

```

You can see how ASP.NET is able to pick up the fields based on your choices. Of course, this gives you a little more control because you change the header text and make the DataGrid more readable to the human eye.

FIGURE 2.8:

Customizing the columns of an XML-based DataGrid



Wrapping Up

Despite its hype, XML isn't the panacea for all your data manipulations issues. It is, in fact, no substitute for good old-fashioned SQL-based relational systems. For now, XML's primary use for ASP.NET-driven sites will probably be for navigation, some content management tasks (including blog sites, which are like Web-based diaries and are currently quite the rage), and Web Services.

If you're looking for very portable solutions, utilize XSLT's standardized vocabulary, since many different environments can work with it. If you know your code will remain in a .NET environment and you are working with a lot of tabular data, consider the DataGrid.

In this chapter you were forced to go into Dreamweaver's Code view to do some coding because Dreamweaver's direct support for XML data sources isn't particularly great.