Web Matrix
Developer's Guide

JOHN PAUL MUELLER

Apress™

Web Matrix Developer's Guide

ISBN (pbk): 1-59059-092-9

Printed and bound in the United States of America 12345678910

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

Web Matrix and XML

In This Chapter

- Use the XML Support That Web Matrix Provides

- Use Microsoft XML Notepad to View XML-Formatted Files

- Create Applications with the XMLEditGrid Control

- Create Applications with the XML File Page

- Create Applications with the XSL Transform Page

- Create Applications with the XML Schema Page

- Design an XML Data Display Application

Everyone's learning to use eXtensible Markup Language (XML) today and employ it in a variety of formats. You'll find XML just about everywhere. Developers use XML to transfer data from one place to another and as part of specifications designed to perform special types of data transfers. It also appears as part of other specifications, such as the Simple Object Access Protocol (SOAP)—see Chapter 8 for details. Some Web sites use XML as a means for displaying data. Many developers have created unique uses for the eXtensible Stylesheet Language (XSL) Transformation (XSLT) in performing data manipulation. A few applications I've seen use XML as an alternative to centralized storage for application settings. You'll even find XML in Database Management Systems (DBMSs). In short, XML is the latest nonplatform, non–language-specific means for storing information.

The fact that you can use XML anywhere and even use it to bridge gaps between computer systems makes it a perfect match for Web Matrix—a tool that I'm convinced will also help you bridge platform gaps. The fact that Web Matrix is lightweight and doesn't perform a lot of interpretation for you means that this product is perfect for working on XML on any platform you choose. The ability to create an FTP connection to manipulate content remotely only makes things better.

This chapter discusses the XML support provided by Web Matrix. As with the material in Chapter 3, this material is decidedly open for anyone to use. You don't have to work on a Microsoft platform or use other Microsoft products to create something with XML. An XML-formatted Web page using XSLT works equally well on an Apache server as it does on Internet Information Server (IIS). You'll find that XML schemas are pretty much universal as well.

We'll discuss two Microsoft-specific topics in this chapter. The first is XMLEditGridControl. This special control helps you create impressive data displays quickly. We'll use this control in the XML data display example application in this chapter. This example provides practical advice on how to implement XML solutions using the XML features of Web Matrix. It also presents you with a specific application of XML that you can use as a source of ideas for your next Web development project.

Understanding XML Support in Web Matrix

The introduction to this chapter says a lot. XML is a text-based data storage technology that relies on tags to separate the various data elements. The formatting required for the tags is relatively straightforward and free form, but the specification includes precise rules you have to follow. For example, every opening tag requires a closing tag, or the document isn't well formed. Web Matrix helps you follow the rules required by the XML specifications, yet simplifies the task of reading and manipulating the data by making the XML classes of the .NET Framework available. (You'll find an overview of the XML classes at `http://msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemXml.asp`.)

This section of the chapter provides an overview of the XML support provided by the .NET Framework, and as a result, by Web Matrix. Microsoft groups the System.Xml namespace with other data-oriented namespaces such as System.OleDb. Essentially, the System.Xml namespace provides classes oriented toward the management and control of data. Many developers think that they'll only use XML for data transfer because of the placement of the System.Xml namespace in the hierarchy, and because of articles that they've read in the trade press and magazines. However, the System.Xml assembly has a lot more to offer than simple data import and export functionality.

XML has a basic structure that lends itself to distributed application development because every computer out there can interpret text. XML relies on a formal use of tags to delineate data elements and format them in a way that two machines can understand. Formatting requires the use of attributes that define the type, scope, and use of the data in question. An XML file usually contains a header that shows the version of XML in use. In some cases, XML also requires special constructs such as the CDATA section to ensure that the recipient interprets the XML-formatted code directly. It's also possible to add comments to an

XML file to help document the content (although the use of comments is relatively uncommon except as a means of identifying the data source). Finally, you need some way to read and write XML to a data stream (be it a file or an Internet connection).

Now that you have some idea of what an XML namespace would have to include, let's look at some class specifics. The following list doesn't tell you about every class within System.Xml, but it does tell you about the classes you'll use most often. We'll use several of these classes in the examples in this chapter.

XmlAttribute and XmlAttributeCollection: Defines one or more data features such as type. Attributes normally appear as part of a database schema or within a Document Type Definition (DTD). Applications need attribute data to convert the text representation of information such as numbers to their locally supported type.

XmlCDataSection: Prevents the XmlReader from interpreting the associated text as tag input. An application could use a CDATA section for a number of purposes, including the transfer of HTML or other tag-based code. CDATA sections are also used with escaped or binary data.

XmlComment: Represents a comment within the XML document. Generally, vendors use comments to include data source information or standards adherence guidelines. However, comments can also document XML data or serve any other human readable text need the developer might have.

XmlDataDocument, XmlDocument, and XmlDocumentFragment: Contains all or part of an XML document. The XmlDataDocument class enables the developer to work with data found in a data set. Data stored using an XmlDataDocument can be retrieved, stored, and manipulated using the same features provided by a data set. Use the XmlDocument to represent the World Wide Web Consortium (W3C) Document Object Model (DOM) that relies on the typical tree representation of hierarchical data. An XmlDocumentFragment represents just a part of an XML document. Developers normally use an object of this class for data insertions into an existing tree structure.

XmlDeclaration: Contains the XML declaration node. The declaration node includes information such as the XML version, encoding level, read-only status, and namespace. An XML document usually contains a single declaration node, but it's possible to use multiple declaration nodes to provide multiple layers of data support.

XmlNode: Represents a single leaf of the XML data hierarchy. An XmlNode usually consists of a single tag pair with associated data (contained within an XmlText object). The XmlNode is the Root object for many other classes in the System.Xml namespace. For example, the XmlDocument and XmlDocumentFragment container classes are derived from XmlNode. At the lower end of the scale, XmlAttribute and XmlEntity are both leaf nodes based on XmlNode. In some cases, developers will use XmlNode directly to parse an XML document.

XmlNodeReader, XmlReader, XmlTextReader, and XmlValidatingReader: Performs a read of XML data from a document, stream, or other source. The XmlReader is the base class for all other readers. This reader provides fast, forward-only access to XML data of any type. The XmlNodeReader reads XML data from XmlNodes only—it doesn't work with schema or DTD data. The XmlTextReader doesn't perform as quickly as other readers, but it does work with DTD and schema data. This reader checks the document and nodes for well-formed XML, but doesn't perform any validation in the interest of speed. Use the XmlValidatingReader when the validity of the data is more important than application speed. This reader does perform DTD, XML-Data Reduced (XDR) schema, and XML Schema Definition (XSD) language schema validation. If either the XmlTextReader or XmlValidatingReader detect an error in the XML, both classes will raise an XmlException.

> **TIP** *Using the correct reader is the most important way to improve application reliability and performance. Using the slow XmlValidatingReader on simple node data ensures your application will perform poorly (much to the consternation of the user). On the other hand, using the XmlNodeReader on mission critical data could result in data loss and unreliable application operation. In fact, due to the need to resend missing or incorrectly resolved data, application performance could suffer as well.*

XmlResolver and XmlUriResolver: Resolves external XML resources pointed to by a Uniform Resource Identifier (URI). For example, many common data types appear as definitions in external, standards-maintained resources. In addition, external resources on your company's Web site, such as a DTD or schema, will also appear in this list. Most developers will use the default resolution capabilities provided by XmlUriResolver. However, you can also create your own resolver using the XmlResolver class as a basis.

XmlTextWriter and XmlWriter: Performs a write of XML data to a data stream, file, or other output. The XmlWriter provides a means of manually controlling the output stream—a requirement in some cases. For example, the XmlWriter provides control over the start and stop of each data element and enables you to declare namespaces manually. However, the XmlTextWriter greatly simplifies the task of outputting data by performing some tasks automatically and making assumptions based on the current application environment.

Working with Microsoft XML Notepad

XML is almost, but not quite, readable by the average human. Reading simple files is almost trivial, but once the data gets nested a few layers deep, reading it can become tiresome. That's why you should have a tool for reading XML in your developer toolkit. The only problem is that some of these tools cost quite a bit for the occasional user. Microsoft has remedied this problem a little with the introduction of XML Notepad (`http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnxml/html/xmlpaddownload.asp`). This utility is free for the price of a download and does a reasonable job of reading most XML files. The current version, 1.5, is in beta as of this writing. In addition, although the Web site doesn't mention it, this version works fine with both Windows 2000 and Windows XP. The following sections provide a brief overview of this tool. We'll use this tool in several sections of the book to read XML data, so you'll want to download a copy.

Opening XML Files

When you start XML Notepad, you'll see a blank project. Use the File ➢ Open command to display an Open dialog box that allows you to open XML files from a local drive or from a Web site. All you need is a filename (and path) or a URL to get started.

TIP *Not all XML files have an XML file extension. We've already seen one case in the book where a CONFIG file was actually an XML file in disguise. Often, you'll find that the file extension matches the customized use of the file, rather than the actual content. When in doubt, try to open a file that looks like it contains XML data to see if XML Notepad will make it easier to understand.*

Figure 7-1 shows the content of a movie database example I created by exporting the MovieGuide database ExistingMovies table to the Movie.XML file. (You'll find several XML files to view in the \Chapter 07\Sample XML Data folder of the source code, available from the Downloads section on the Apress site at `http://www.apress.com/book/download.html`.) Notice that the name of the elements matches the name of the table for the movie database (found in the Movie.XML file). Likewise, each of the child elements matches the name of one of the fields within the table. The right pane shows the data contained within each one of the child elements.
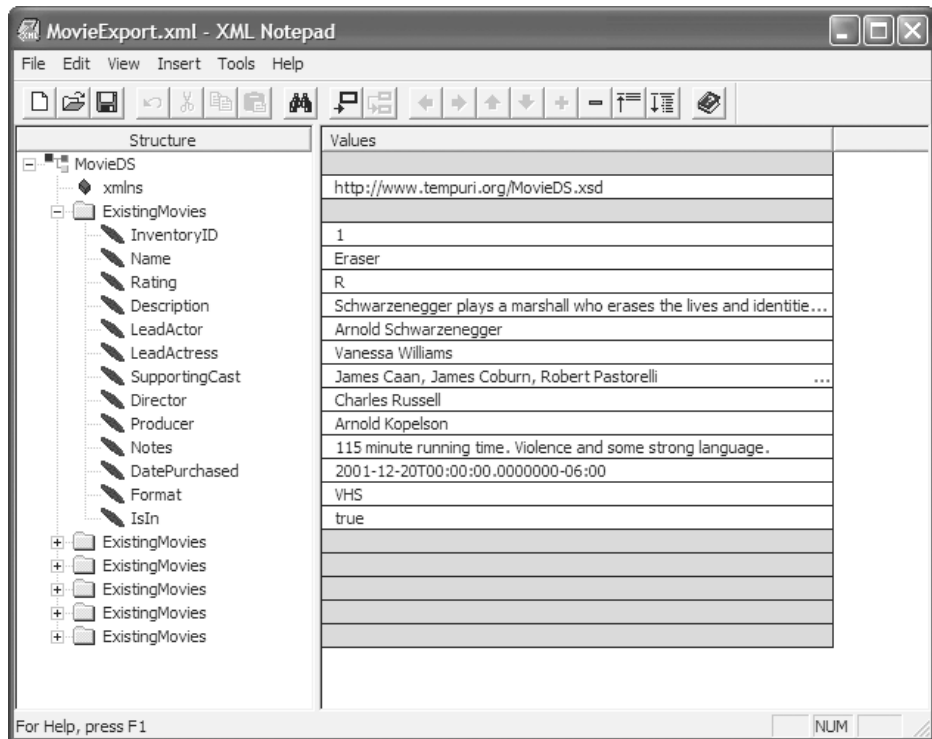

Figure 7-1. The names of the elements are important when working with exported data in XML format.

> **TIP** *If you want to use XML from your SQL Server 2000 installation, then you'll want to download the XML feature packs found at* `http://msdn.microsoft.com/downloads/sample.asp?url=/MSDN-FILES/027/001/824/msdncompositedoc.xml`. *These feature packs add to the capabilities already found in SQL Server 2000. In addition, they help you keep your setup current by ensuring that your SQL Server 2000 installation always meets the specification requirements.*

Modifying Existing Data and Creating New Data

You can use XML Notepad for more than just viewing data. Creating new data for testing purposes is relatively painless once you see the exported data from an existing database. In fact, we'll create some sample data in this section for use in an application later. The following steps will get you started.

1. Create a blank project using the File ➢ New command.

2. Type the name of the data set in the Root object. The example uses SampleData as a name, but you'll want to select something a little more descriptive for a production application.

3. Rename the first element to reflect the new table. The example application uses NewDataElement as the name. DBMSs normally use the database name for the Root object and the table name for the first element. (See Figure 7-1 for an example of actual database output.)

4. Add a new child element using the options on the Insert menu. You'll notice that the first element changes into a folder. Type the name of the first data column in this element. The example uses Data1 for the first child element. However, a database would use the name of the first field in the selected table (see Figure 7-1).

5. Add additional columns as needed until you complete one record's worth of entries. The example uses Data2 and Data3 as entries to complete one record in the sample database, but you should also look at Figure 7-1 as another example of what you could do.

6. Type values for each of the child elements. Now that you have one complete record, you can use the Duplicate command to create copies of it. Each copy will become one record within the XML database.

7. Right-click the NewDataElement folder and choose Duplicate from the context menu. The example provides three records for the sample, but you can include any number you wish. Figure 7-2 shows the structure and contents of the SampleData.XML file. It also shows the content we'll use for the example.
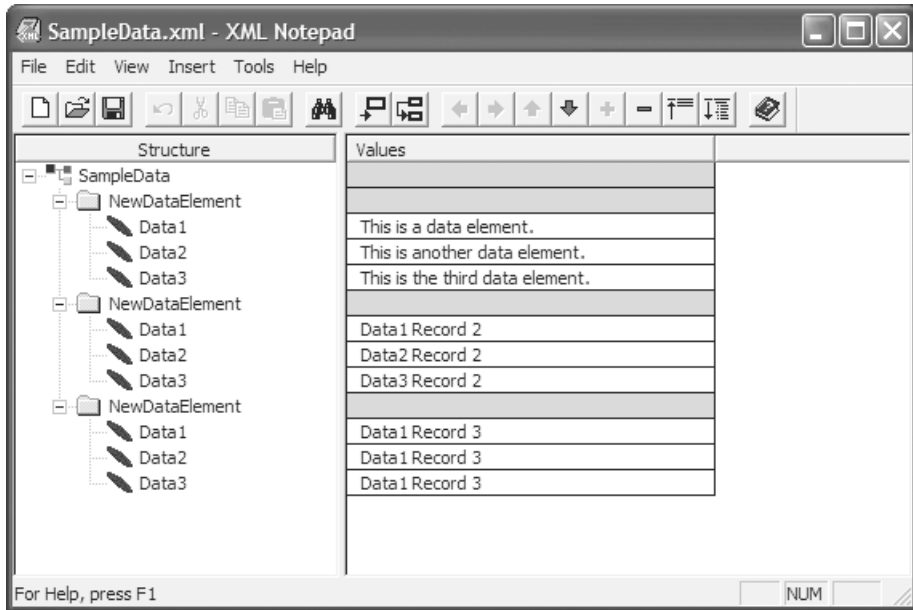
Figure 7-2. Creating an XML database using XML Notepad is relatively easy as long as you follow a few rules.

Now that you've seen what XML Notepad can do, you need to realize that other vendors do provide other alternatives. XML Notepad doesn't have some of the bells and whistles of high-end products such as XML Spy (`http://www.xmlspy.com/`). However, XML Notepad is a good alternative if you only use an XML editor occasionally and don't want to spend any money. The important consideration is that you have an XML editor that you can use to view the output from your applications.

Using the `XmlEditGrid` Control

Just in case you've forgotten, the XmlEditGrid control isn't part of the usual assortment of Web Matrix controls. This control is the one we downloaded from the Online Component Gallery in the "Connecting to the Online Component Gallery" section of Chapter 2. If you haven't downloaded this control already, you'll want to download it using the instructions in Chapter 2 before you begin working with this section of the chapter.

> **TIP**   *Don't get the idea that the Online Component Gallery is the only place to get new controls for your Web Matrix setup. The online community has worked hard to put together an impressive list of controls that you can use with Web Matrix. In some cases, the controls aren't well documented, but they're normally based on existing (documented) controls and consequently easy to figure out. See the Control Gallery at* `http://www.asp.net/Default.aspx?tabindex=2&tabid=30` *for details.*

The XmlEditGrid control is unique in that it treats correctly formatted XML files as database input. Of course, the key phrase is "correctly formatted"—the XML file must use a format that lends itself to display within a grid. The SampleData.XML file we discussed earlier does have the proper format for display in the XmlEditGrid, so we'll use it in this example. Other files in the Sample XML Data folder will also work, but this file is particularly easy to understand.

You can use the XmlDataGrid as you would any other grid for database work. The XmlDataGrid includes features that help you edit, delete, and add records to the XML file using an interface that looks just like a standard database grid. Figure 7-3 shows an example of the XmlDataGrid loaded with the movie database (found in Movie.XML) we looked at earlier in the chapter.

Figure 7-3. Use the XmlDataGrid to view XML files as you would any database file.

Setting the XmlEditGrid up for use is easy. Begin by adding the control to the form. If you want to use this control for XML files, you must supply the name of a file in the XmlFile property. The XmlEditGrid can also use a data source employing techniques similar to the ones we examined in Chapter 5. In short, the XmlEditGrid is a data grid with some extra features.

I found that some configuration tasks are easier if you open the properties dialog box shown in Figure 7-4 by clicking the Property Pages icon in the Properties window. The dialog box groups some task elements, such as formatting, in an easy to understand manner. You can still use the Properties windows, but the dialog box presents fewer problems. The default setup for this control uses almost no formatting at all, making it difficult to see some features, such as the currently selected record, so you'll definitely want to perform some configuration.
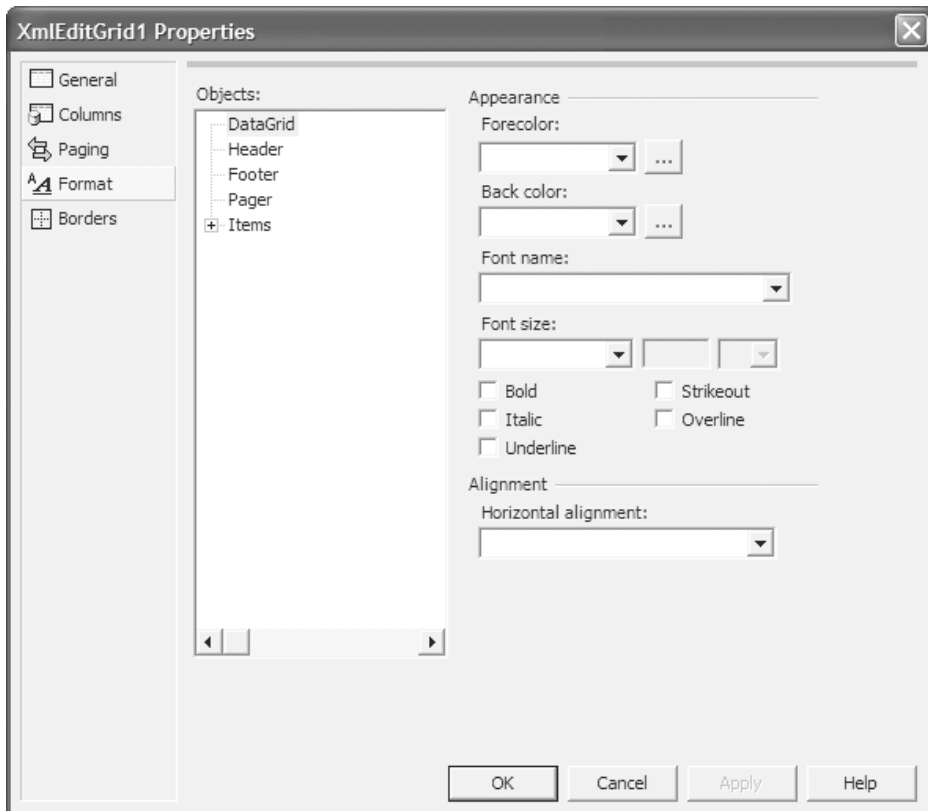

Figure 7-4. Using this dialog box to perform tasks such as formatting makes the job easier.

Depending on how you configure the XmlEditGrid, it will automatically provide basic editing functionality for XML files (I didn't test the database capability because we have other controls to use for that purpose). You can edit, delete, and add records to your XML file database. The XmlEditGrid provides an edit icon in the leftmost column (looks like a pencil). Click this icon the first time, and the XmlEditGrid will select the row for editing—click it the second time, and the changes appear in the file and the control deselects the record. Immediately to the right of this icon is the delete icon (looks like an X). The control doesn't provide any warning about deleting a record—it simply performs the act on the selected row when clicked. The add feature appears as the Add new item link at the bottom of the grid. You can override most of the default actions performed by the XmlEditGrid by creating event handlers for them.

You'll find that there are a couple of idiosyncrasies with this control. The first is that when you install the control from the Web, it will usually register itself in the GAC on your local machine. (If you don't register it in the GAC, make sure you create a \bin folder for each application and place the control there.) Unfortunately, when you move the application to your server for testing, it will fail. You'll find the control located in the \Program Files\Microsoft ASP.NET Web Matrix\v0.5.464\Components folder of your machine. Copy the control to the server and use GACUtil to register it. The control location doesn't matter, as long as you properly register the control—CLR will find it.

The second problem is a little harder to figure out. Figure 7-5 shows the display you'll see when you get the application running the first time. Notice that all of the icons are missing. This problem doesn't prevent the application from working, but it's an annoying problem from an appearance perspective.
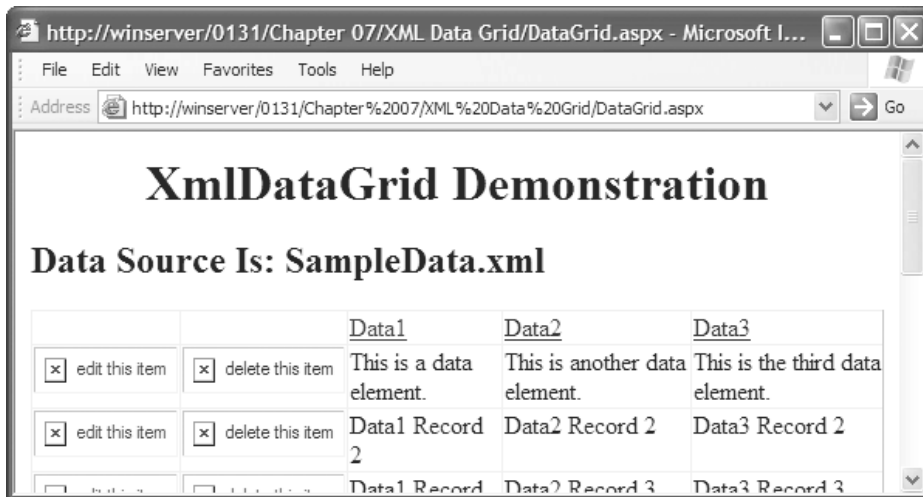

Figure 7-5. Fixing this error can prove bothersome until you know how.

The way to fix this problem is to use the View ➢ Source command in Internet Explorer to view the source code that the control creates. You'll notice that every entry has this bit of code.

```
<img src='/aspnet_client/Swarren_XmlEditGrid/1_0_0_0/edit.gif'
```

The only problem is that you don't have the required folder on your server, nor did the control tell you where to find the image on your local machine. You can find the errant icons in the \Program Files\ASP.NET\Client Files\Swarren_XmlEditGrid\1_0_0_0 folder of your local hard drive.

It seems that the installation program assumes that you'll use the local Web server for all testing. Simply create an \aspnet_client\Swarren_XmlEditGrid\1_0_0_0 folder in the root directory of your IIS installation, and copy the GIF files to it. The icons will mysteriously reappear in your application. Of course, this opens the door for customization. If you don't particularly like the icons that the XmlEditGrid uses, replace the GIF files in this folder with icons that you do like—just make sure you use the correct names.

The XmlEditGrid control performs a lot of the work for you right out of the package. Even so, I needed to add two pieces of code to the example. The first displays the name of the XML file, while the second allows for sorting. The one feature that doesn't appear to work correctly right out of the box is sorting. Listing 7-1 shows how to add both features to your application.

Listing 7-1. Code for Sorting Grid and Display Filename

```
void Page_Load(Object sender, EventArgs e)
{
    // Display the name of the XML file.
    Label1.Text = "Data Source Is: " + XmlEditGrid1.XmlFile;
}


void XmlEditGrid1_SortCommand(Object source, DataGridSortCommandEventArgs e)
{
    // Define a sort order for the XmlEditGrid.
    Response.SetCookie(new HttpCookie("SortOrder", e.SortExpression));

    // Tell the user about the sort order.
    Label2.Text = "The sort order is: " + e.SortExpression;
}
```

As you can see, the application adds the filename during the page loading process, which means you could allow editing of more than one XML file using the same page without losing the user. The `XmlEditGrid1_SortCommand()` uses the same technique the DataGrid control uses for sorting—you send a cookie back to the server with the required sort order. This method also adds text to a second label that shows the sort order to the user. The XmlEditGrid is unsorted when you first display the data. Figure 7-6 shows the final output of this example using the SampleData.XML file.
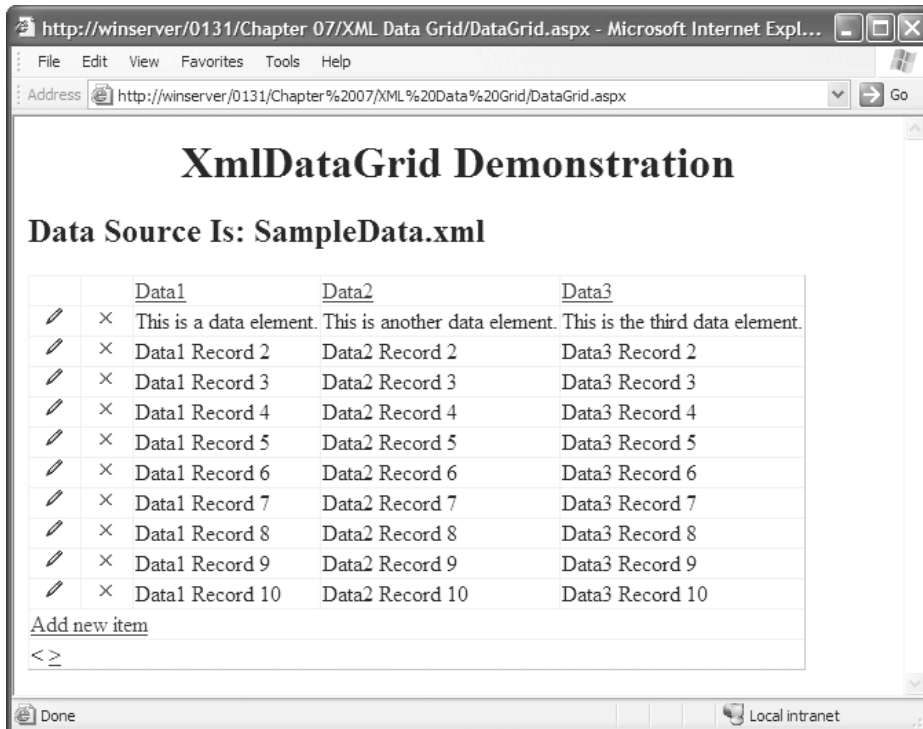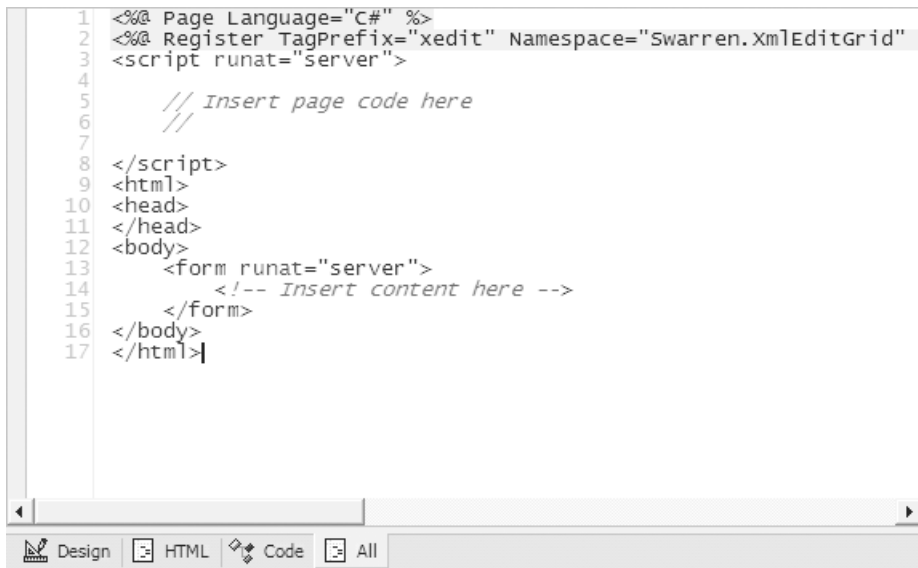

Figure 7-6. Creating output with any XML file that contains data in the correct format is easy with XmlEditGrid.

Removing Custom Controls from Your Project

There's a small, but important, change that occurs when you add a custom control to your project—one that you downloaded from the Internet or otherwise added to your toolbox. Web Matrix automatically adds a `<%@ Register>` tag to your code. As we saw in Chapter 6, this tag registers the control for you and makes it possible for ASP.NET to load it. Consequently, adding the tag is a good service for Web Matrix to provide.

Unfortunately, as shown in the following illustration, this tag is only visible when you view the All tab of the editing window. For example, I added the XmlDataGrid control to this project. You can see the `<%@ Register>` tag immediately below the `<%@ Page>` tag.

```
1   <%@ Page Language="C#" %>
2   <%@ Register TagPrefix="xedit" Namespace="Swarren.XmlEditGrid"
3   <script runat="server">
4
5       // Insert page code here
6       //
7
8   </script>
9   <html>
10  <head>
11  </head>
12  <body>
13      <form runat="server">
14          <!-- Insert content here -->
15      </form>
16  </body>
17  </html>
```

Design | HTML | Code | All

When you remove a custom control from the Design tab, the `<%@ Register>` tag remains in place. Unless you happen to look at the All tab, you won't see the tag and could try to deploy your application with the `<%@ Register>` tag in place. If you don't include the requisite DLL with your application, it won't run even if the code is correct. Removing the tag will fix the problem.

Using the XML File

Let me begin by reiterating that using Web Matrix isn't the best way to create an XML file unless you like to do a lot of typing and don't want to see what your data actually looks like. That said, the ability to create XML files is important. When you use the XML File page template, you get a basic tag that starts you on your way. All you need to do is begin typing the elements and attributes for your XML file.

> **TIP** *Web Matrix provides a text editor for the XML files. This might seem problematic at first, but I've found it's quite useful. Most XML editors on the market, including XML Notepad, operate on the premise that the XML file you want to edit has "well-formed" XML. Some XML files won't meet this requirement in certain situations, such as some CONFIG files I've viewed. The fact that Web Matrix provides text-editing functionality means you can view the errant XML file and fix it if required. The point is that Web Matrix provides you with another alternative—another tool for your toolkit.*

This section looks at a relatively simple example. Learning to type XML by hand is actually a good idea because it helps you see problems in automatically generated XML later. Some of the elusive issues behind a relatively simple packaging technique begin to make sense as you spend more time working with XML. With this in mind, consider the XML code in Listing 7-2.

Listing 7-2. A Simple XML File

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<PhoneMessages>
    <Date Day="09/16/02">
        <Message Name="George">
            "Hello, Sorry I missed you!"
        </Message>
        <Message Name="Sam">
            "Are you gone again?"
        </Message>
    </Date>
    <Date Day="09/17/02">
        <Message Name="Ann">
            "Want to go fishing Thursday?"
        </Message>
    </Date>
```

```
        <Date Day="09/18/02">
            <Message Name="Jerry">
                "Hope to see you tomorrow!"
            </Message>
            <Message Name="Samantha">
                "Call the office ASAP!"
            </Message>
            <Message Name="Jan">
                "Give me a call at work!"
            </Message>
        </Date>
</PhoneMessages>
```

This XML is well formed because it includes a header, and each of the elements has an opening and closing tag. Each element contains either another element or at least one attribute. The attributes take two different forms. You give the attribute a name and place it within the tag, or your can place it between an opening and closing tag. The specification doesn't require indenting. However, most developers find it useful for reading the XML in this format. Figure 7-7 shows how this XML file looks in XML Notepad.
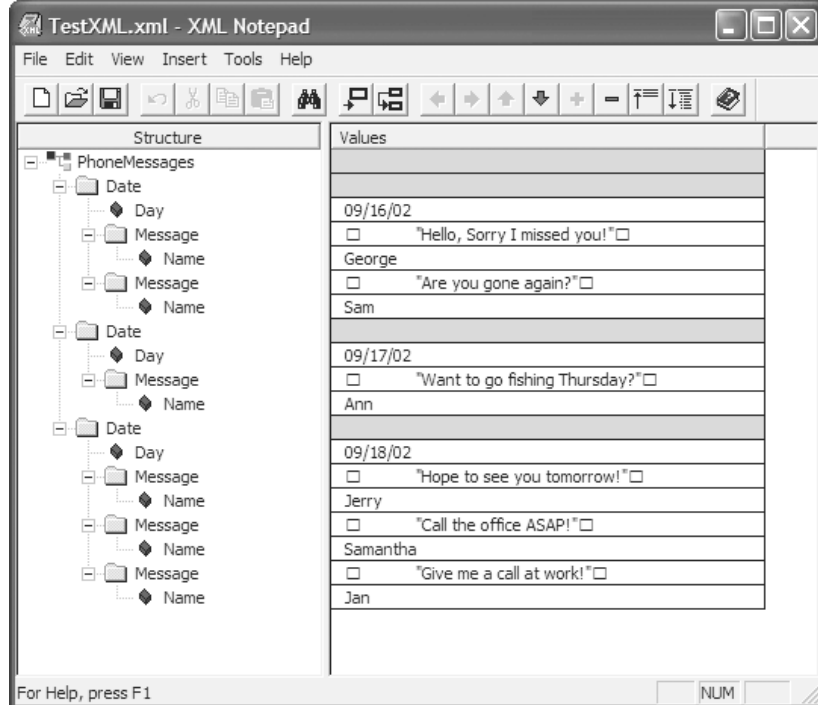

Figure 7-7. Writing XML by hand can reward you with a better understanding of how this data storage format works.

Using the XSL Transform

Defining methods for reading XML must occupy more than a few passing thoughts for some developers because the developer community has created a number of ways to present XML in a readable format. The XSL Transform page provides another text-oriented method for working with what appears as XML. However, XSLT files can contain more than just XML, and browsers don't use them for direct display. An XSLT file contains formatting information—a means for displaying the content of an XML file in a form that the average human can read. In addition, the combination of an XML and an XSLT file can create a presentation in a standard browser.

The XML file provides content, while the XSLT file provides format. You can take the same content and present it in a number of ways by changing XSLT files. This feature makes it possible for some Web sites to provide a standard and printer friendly view of the same data. The data hasn't changed—only the XSLT file content has changed.

Let's begin with the one and only change you have to make to the XML file. To use an XSLT file, you need to add a tag to the beginning of the XML file. This tag tells the browser which XSLT file to use to format the data contained within the XML file. If the browser doesn't find the XSLT file, it will still display the data using whatever method it normally uses for displaying XML files, which isn't very readable by the average user.

```
<?xml-stylesheet type="text/xsl" href="PFormat.XSLT"?>
```

As you can see, the tag simply says that this XML file uses a stylesheet. The type of information in the stylesheet file is a combination of text and XSL. Finally, the name of the XSLT file is PFormat.XSLT. These three entries are all you need to change the way the browser displays the XML file.

Creating an XSLT file is a little more complicated. Listing 7-3 shows one of two XSLT files we'll discuss in this section.

Listing 7-3. A Simple XSLT File

```
<?xml version='1.0'?>
<xsl:stylesheet version='1.0'
xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
<xsl:output method="xml" indent="yes" />
<xsl:template match="/">

<!-- Create the HTML Code for this stylesheet. -->
<HTML>
```

```
<HEAD>
    <TITLE>Remote Performance Monitor</TITLE>
</HEAD>

<BODY>
<CENTER><H3>Performance Monitor Results</H3></CENTER>

<TABLE BORDER="2">
    <TR>
        <TH>Time</TH>
        <TH>Percent User Time Value</TH>
    </TR>
        <xsl:apply-templates select="//Item"/>
</TABLE>

</BODY>
</HTML>
</xsl:template>

<!-- XSL template section that describes table content. -->

<xsl:template match="Item">
    <TR>
        <TD>
            <xsl:value-of select="Time"/>
        </TD>
        <TD>
            <xsl:value-of select="Data"/>
        </TD>
    </TR>
</xsl:template>

</xsl:stylesheet>
```

The code begins with a heading. Always include the XML heading because this is essentially a form of XML. The next tag defines this file as a stylesheet, while the third tag defines the method of output. The fourth tag is especially important because it defines which tags this stylesheet matches in the source XML file. In this case, the stylesheet matches (or imports) all of the tags.
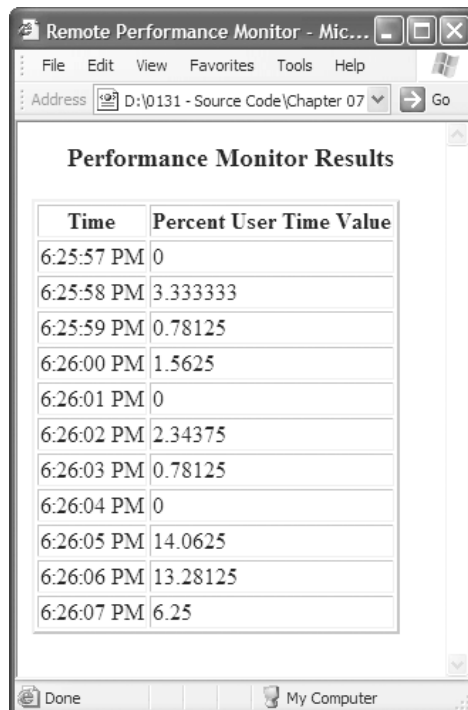
The next bit of code begins defining the HTML output for this example. The code is purposely simple in this case—it won't even pass the W3C test because it lacks features such as the <DOCTYPE> tag. However, the HTML output will display

a result in a browser, which is all we need now. Nothing prevents you from creating relatively complex HTML using an XSLT file.

After the code defines a header, it begins defining the body of the page. In this case, we'll use a table. However, now you need to understand something about the XML file we're using. The target XML file contains entries in this format.

```
<Item>
  <Time>6:25:57 PM</Time>
  <Data>0</Data>
</Item>
```

The file places each entry within an `<Item>` element. The subelements, `<Time>` and `<Data>`, include the information we want to display. Look at the code and you'll see an `<xsl:apply-templates select="//Item"/>` entry. This entry tells the browser to display each entry in the XML file according to the definitions provided in the XSL template that matches the `<Item>` element. The template begins after the code completes the HTML portion of the information with the `<xsl:template match="Item">` tag. At this point, we select the values contained in each of the subelements and display them on screen using standard HTML. Figure 7-8 shows the output of the XML file and XSLT file combination.

Figure 7-8. Reading this output is easier than viewing straight XML.

Let's look at a more complex XSLT file. Remember that the data hasn't changed—only the content of the template has changed. In this case, the view will add a counter, so we know which entry we're looking at, and use a two-column format in place of the single-column format shown in Figure 7-8. Listing 7-4 shows the code we'll use in this case.

Listing 7-4. A Better XSLT File

```
<?xml version='1.0'?>
<xsl:stylesheet version='1.0'
xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
<xsl:output method="xml" indent="yes" />
<xsl:template match="/">

<!-- Create the HTML Code for this stylesheet. -->
<HTML>
<HEAD>
    <TITLE>Another Performance Monitor View</TITLE>
</HEAD>

<BODY>
<H3 ALIGN="Center">Performance Monitor Results</H3>
<P ALIGN="Center">Version 2</P>

<TABLE BORDER="2">
    <TR bgcolor="#C0C0FF">
        <TH>Entry</TH>
        <TH>Time</TH>
        <TH>Percent User Time Value</TH>
        <TH>Entry</TH>
        <TH>Time</TH>
        <TH>Percent User Time Value</TH>
    </TR>
    <xsl:apply-templates select="//Item"/>
</TABLE>

</BODY>
</HTML>
</xsl:template>

<!-- XSL template section that describes table content. -->
```

```
<xsl:template match="Item">
    <xsl:if test="position() mod 2 = 1">
        <xsl:text disable-output-escaping = "yes">
            &lt;TR&gt;
        </xsl:text>
    </xsl:if>
        <TD>
            <xsl:value-of select="position()"/>
        </TD>
        <TD>
            <xsl:value-of select="Time"/>
        </TD>
        <TD>
            <xsl:value-of select="Data"/>
        </TD>
    <xsl:if test="position() mod 2 = 0">
        <xsl:text disable-output-escaping = "yes">
            &lt;/TR&gt;
        </xsl:text>
    </xsl:if>
</xsl:template>

</xsl:stylesheet>
```

A lot of this code is the same as before—at least the layout of the code is the same. We still have an HTML section and a template section that interprets the data. As you can see, the HTML section includes a new header and then doubles the headers so we can see two rows. I added a little color to the table header for this example—it's a nice touch to differentiate between the two areas.
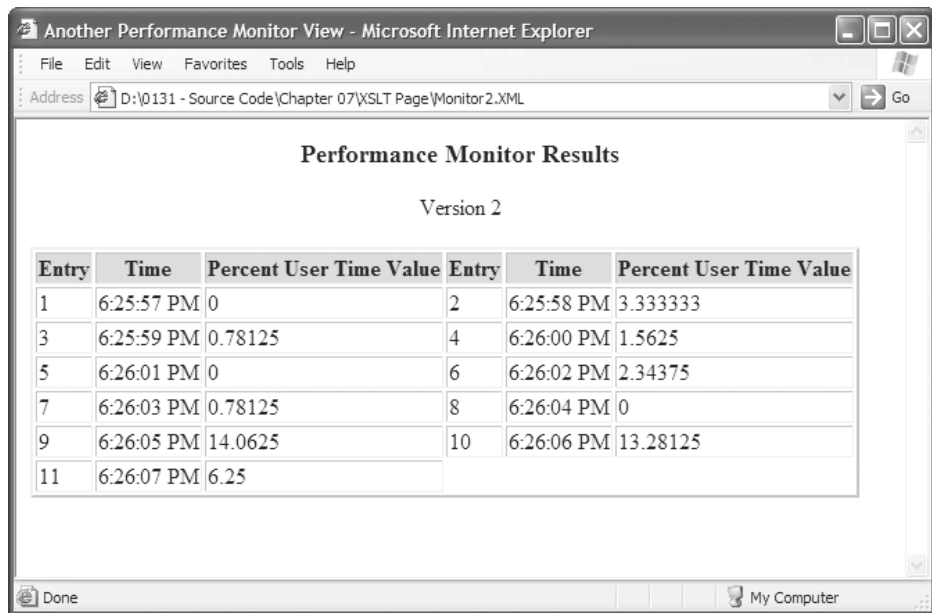
> **TIP**   *XSLT code might look a little intimidating at first, but it's relatively straightforward once you work with it for a while. You'll find an excellent XSLT reference and examples at* `http://www.zvon.org/xxl/XSLTreference/Output/index.html`. *The examples are a little abstract—they aren't HTML specific, but that's actually a good feature because you can use XSLT for more tasks than we are doing here. Developers use XSLT for a number of translation tasks.*

The template uses a few more XSLT features. The first `<xsl:if>` tag performs a test on the current position within the file. If the entry is an odd value, then it executes the code that follows. Otherwise, execution continues with the next step, which is to display one of the entries. The code that this first `<xsl:if>` tag executes is to print out some text using the `<xsl:text disable-output-escaping = "yes">` tag. Using this tag permits you to output a `<TR>` tag to the browser. However, the tag only appears on screen when the template is processing an odd entry.

This version of the template will output an item number. You still use the `<xsl:value-of>` tag. However, instead of selecting the output of an entry in the XML file, this tag selects the output of the `position()` function. The `position()` function comes in very handy in processing XML files because it tells you where you are—it's the equivalent of the record number for a database table.

The final change for this XSLT file is to add another `<xsl:if>` tag. Instead of checking for odd entries, this one checks for even entries. As with the first `<xsl:if>` tag, this one outputs text in the form of the closing `</TR>` tag. Because of the arrangement of `<xsl:if>` tags, the display will show two XML file entries in each row. Figure 7-9 shows the output of this example.

Figure 7-9. Creating a different look means changing the XSLT file, not the XML data.

Using the XML Schema

Databases require a schema to define their content. The tables within the database have columns that contain specific types of data. Each entry has a name, such as LastName, for the last name entry in a contact management database. The entry has a data type such as text, a length, and other characteristics that define that entry. An application relies on these entries to interact with the database. For example, the entry will tell the application whether an edited entry is too long or of the wrong type. All of this information helps create a reliable flow of information between the database and the application.

> **TIP** *Most DBMSs will create XSD files for you. However, if you find that you have to create such a file by hand, make sure you use the standards-recognized method. One of the best places to learn about the XSD file format is the XML Schema page on the W3C site at* `http://www.w3.org/XML/Schema.` *You can find a few coding examples on the Got Dot Net site at* `http://samples.gotdotnet.com/quickstart/howto/doc/Xml/XmlReadWriteSchema.aspx` *that will help you understand how XSD works.*

In order for XML applications to provide the same level of support that their desktop counterparts do, they need some form of schema to use with the XML data they receive from a remote location. The XSD file contains the information that a desktop application would normally obtain using the database schema. As with everything else in this chapter, the data appears in XML format. In fact, many of the XML editors on the market also provide direct support for XSD files. Figure 7-10 shows an example of an XSD file that I created for the Movie.XML file mentioned elsewhere in this chapter. (You'll probably want to open a copy of this file in XML Notepad as well.)
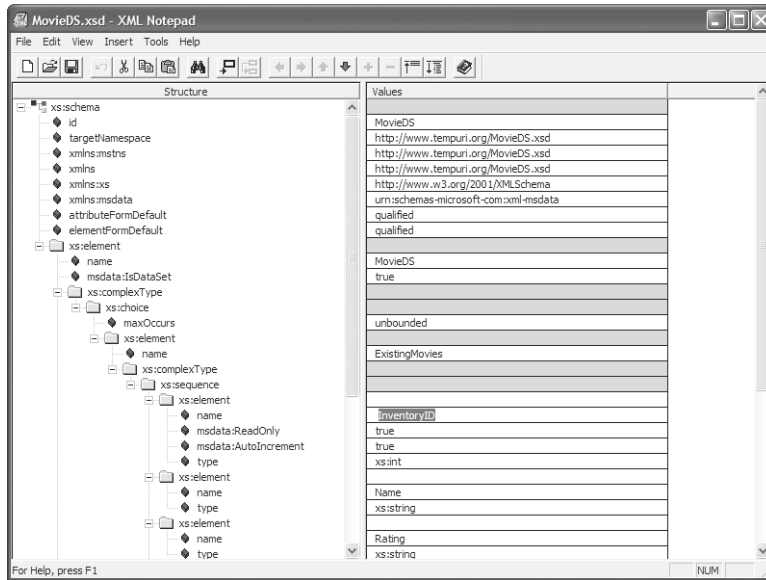
Figure 7-10. Create an XSD file such as this one to describe the format of any table you produce.

We won't look at the source code for this file because it's relatively large. However, you should notice a few things in the XML Notepad display. For example, you'll find a number of namespace entries at the top of the file. Each `<xlmns>` tag defines a particular namespace used within the file or by applications. In a few cases, the application will ignore the namespace for now, but might use it in the future.

Immediately below the headers, we begin creating the data set. The first element is the data set name, MovieDS. The file contains some settings for the data set and then names the first (and only) table, ExistingMovies. Beneath the table entry are some more settings and, finally, the field names, each with their settings. In short, this is a complete description of a data set used to access the ExistingMovies table in a database.

Now that you've seen a schema for a data set, let's concentrate on something a little simpler. As previously mentioned, creating an XML Schema page will generate an XSD file with the opening and close tags, along with a few namespace entries. You have to generate all of the other code required to define the schema you want to create. Listing 7-5 shows a simple XSD file that defines a schema for a contact entry database.

Listing 7-5. A Simple XSD File

```xml
<?xml version="1.0" encoding="utf-8" ?>
<xsd:schema targetNamespace="http://tempuri.org/Simple.xsd"
  xmlns="http://tempuri.org/Simple.xsd"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:annotation>
    <xsd:documentation xml:lang="en">
        A simple XSD file used to demonstrate schemas.
        This is a contact management database.
    </xsd:documentation>
</xsd:annotation>

<xsd:element name="ADatabase">
    <xsd:complexType>
        <xsd:element name="MyContacts">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="name"      type="xsd:string"/>
                    <xsd:element name="street"    type="xsd:string"/>
                    <xsd:element name="city"      type="xsd:string"/>
                    <xsd:element name="state"     type="xsd:string"/>
                    <xsd:element name="zip"       type="xsd:string"/>
                    <xsd:element name="telephone" type="xsd:string"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:complexType>
</xsd:element>

</xsd:schema>
```

This example is simplified, but it demonstrates essential features a schema would require. You need to define the relevant namespaces to begin. Adding some annotation helps anyone who uses the schema you create. Finally, the main section of this schema describes the database. The database name is ADatabase. It contains a single table called MyContacts. The MyContracts table includes a number of text fields that describe the contacts for this person. Figure 7-11 shows how this schema appears in XML Notepad.
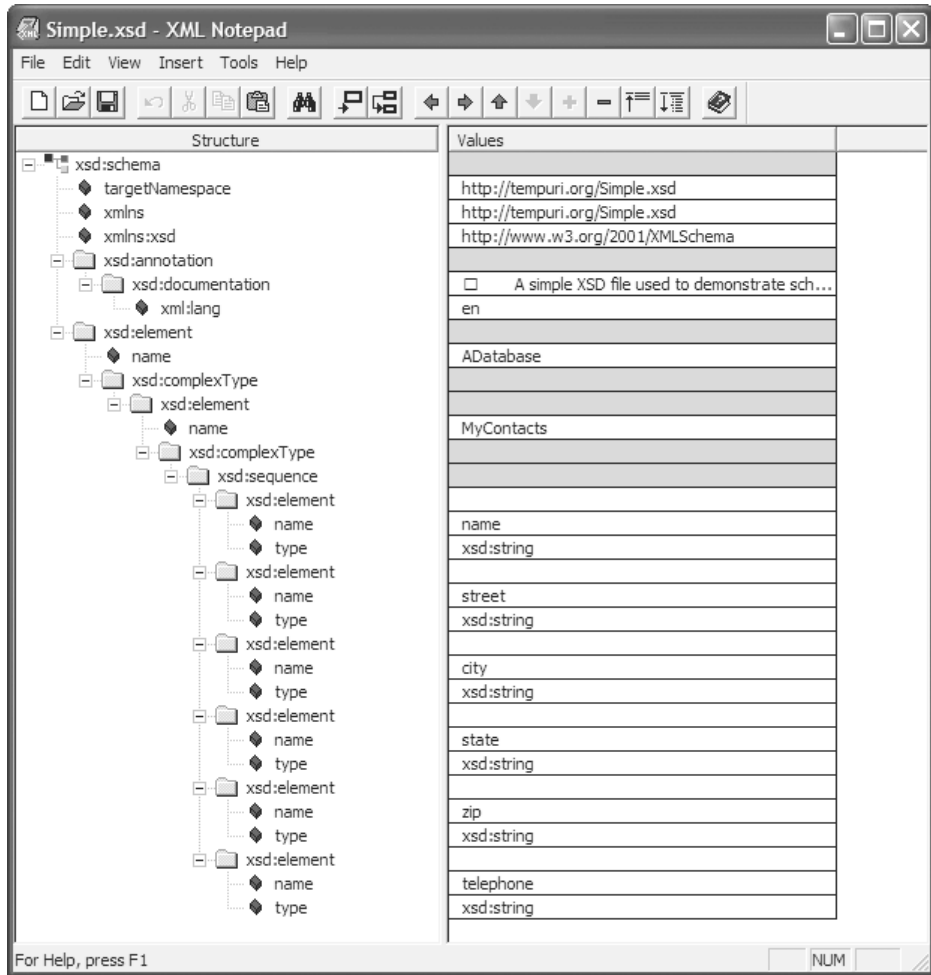
Figure 7-11. Defining even a simple schema can become code intensive.

Summary

This chapter has shown you how to work with the XML features that Web
Matrix provides. More importantly, you've learned one more way in which
Web Matrix helps you bridge the gap between platforms and diverse technolo-
gies. Of course, you also learned about two Microsoft-specific XML implemen-
tations, the XMLEditGrid control and a technique for using it to display XML-
formatted data.

Now it's time to work with some XML data of your own. Look at the various
ways that you can use Web Matrix to bridge gaps in your own organization. You'll
also want to download XML Notepad to work with the data files created by Web

Matrix and other XML applications. We only touched on some of the more important features that XML Notepad provides in this chapter. I think you'll find this utility is an essential part of any toolkit designed to work with XML and its derivatives.

Chapter 8 is almost a continuation of this chapter. However, we move beyond mere XML into the world of Web services—a special means of sharing code and data with other companies. Web service applications have a lot of potential that's unrealized now because vendors still have some problems to work out with it (such as a secure method for transferring requests). This chapter will contain a few surprises for many of you because recent developments have made Web services a viable way to work with distributed applications.