VB.NET Developer's Guide to ASP.NET, XML, and ADO.NET

Chris Kinsman and Jeffrey McManus

DRAFT

SAMS

201 West 103rd St., Indianapolis, Indiana, 46290 USA

0-672-32131-9
Summer 2001

DRAFT

CHAPTER

11

Using XML

IN THIS CHAPTER

- Advantages of XML 5   5
- XML Document Structure and Syntax   6
- Accessing XML Data   11
- Creating XSD Schemas   27
- Class Reference   27

Here's a problem you've probably faced before. A customer or colleague comes to you asking for help working with an application that was written five years ago. Nobody who originally worked on the application still works for the company; the original developer died in a bizarre gardening accident some years back. The customer wants you to write a Web-based reporting system to handle the data emitted by this dinosaur application.

You now have the unenviable task of figuring out how this thing works; parsing the data it emits; and arranging that data in some recognizable format—a report.

Let's assume that the developer of the original application attempted to make it easy on you by expressing the data in some standardized format. A common format is one in which elements within rows of data are separated from each other by a designated character, like a comma or a tab. This is known as a *delimited* format. The following listing demonstrates a comma-delimited document:

```
Jones,Machine Gun,401.32,New York
Janson,Hand Grenade,79.95,Tuscaloosa
Newton,Artillery Cannon,72.43,Paducah
```

But there are a few problems with the delimited format. First of all, what happens if the data itself contains a comma or a tab? In this case, you're forced to use a more complicated delimiter, typically a comma with data enclosed in quotation marks. The fact that different documents can use different delimiters is a problem in itself, though. There's no such thing as a single universal parse algorithm for delimited documents.

To make it even more difficult, different operating systems have different ideas about what constitutes the end of a line. Some systems (like Windows) terminate a line with a carriage return and a line feed (ASCII 13 and 10, respectively), while others (such as Unix) just use a line feed.

Another problem: What *is* this data? Some of it, like the customer's name and the item, is obvious. But what does the number "401.32" represent? Ideally we want a document that is *self-describing*—one that tells us at a glance what all the data represents (or at least gives us a hint).

A third big problem with delimited documents: How can you represent related data? For example, it might be nice to be able to see all the information about customers and orders in the same document. You can do this with a delimited document, but it can be awkward. And if you've written a parser that expects four fields and you suddenly bring in six more related fields between the customer name and the product name, you've broken your parser.

Internet technology mavens realized that this scenario is frighteningly common in the world of software development, particularly in Internet development. XML was designed to replace delimited data (as well as other data formats) with something standard, easy to use and understand, and powerful.

Advantages of XML

In a net application, interoperability between various operating systems is crucial; the transfer of data from point A to point B in a standard, understandable way is what it's all about. For tasks that involve parsing data, then, using XML means spending less time worrying about the details of the parser itself and more time working on the application.

Here are some specific advantages of XML over other data formats:

- *Documents are easily readable and self-describing*—Like HTML, an XML document contains tags that indicate what each type of data is. With good document design, it should be reasonably simple for a person to look at an XML document and say, "this contains customers, orders and prices."
- *XML is interoperable*—There's nothing about XML that ties it to any particular operating system or underlying technology. You don't have to ask anyone's permission or pay anyone money to use XML. If the computer you're working on has a text editor, you can use it to create an XML document. Several types of XML parsers exist for virtually every operating system in use today (even really weird ones).
- *XML Documents can be hierarchical*—It's easy to add related data to a node in an XML document without making the document unwieldy.
- *You don't have to write the parser*—There are several types of object-based parser components available for XML. XML parsers work the same way on virtually every platform. The .NET platform contains support for the Internet-standard XML Document Object Model (DOM), but Microsoft has also thrown in a few XML parsing widgets that are easier to use and perform better than the XML DOM; we'll cover these later in this chapter.
- *Changes to your document won't break the parser*—Assuming that the XML you write is syntactically correct, you can add elements to your data structures without breaking backward compatibility with earlier versions of your application.

Is XML the universal panacea to every problem faced by software developers? XML won't wash your car or take out the garbage for you, but for many tasks that involve data, it's a good choice.

At the same time, Visual Studio.NET hides much of the implementation detail from you. Relational data in the form of XML is abstracted in the form of a DataSet object. XML schemas (a document that defines data types and relationships in XML) can be created visually, without writing code. In fact, VS.NET can generate XML schemas for you automatically by inspecting an existing database structure.

So why learn XML? In the .NET framework, XML is very important. It serves as the foundation for many of the .NET technologies. Database access is XML-based in ADO.NET. Remote interoperability, known as Web Services or SOAP, is also XML-based. It is true that many of the implementation details of XML are hidden inside objects or inside the Visual Studio.NET development environment. But for tasks like debugging, interoperability with other platforms, performance analysis and your own peace of mind, it still makes sense for a .NET developer to have a handle on what XML is, how it works and how it is implemented in the .NET framework.

XML Document Structure and Syntax

XML documents must adhere to a standard syntax so that automated parsers can read them. Fortunately, the syntax is pretty simple to understand, especially if you've developed Web pages in HTML. The XML syntax is a bit more rigorous than that of HTML, but as you'll see, that's a good thing. There are a million ways to put together a bogus, sloppy HTML document, but the structure required by XML means that you get a higher level of consistency.

Declaration

The XML *declaration* is the same for all XML documents. Following is an XML declaration:

```
<?xml version="1.0"?>
```

The declaration says two things: This is an XML document, and this document conforms to the XML 1.0 W3C recommendation (which you can get straight from the horse's mouth at `http://www.w3.org/TR/REC-xml`). The current and only W3C recommendation for XML is version 1.0, so you shouldn't see an XML declaration that's different from this example—but you might in the future as the specification is revised into new versions.

NOTE

A W3C recommendation isn't quite the same as a bona fide internet standard, but it's close enough for our purposes.

The XML declaration, when it exists, must exist on the first line of the document. The declaration does not *have* to exist, however—it is an optional part of an XML document. The idea behind a declaration is that you may have some automated tool that trawls document folders looking for XML. If your XML files contain declarations, it'll be much easier for such an automated process to locate XML documents (as well as differentiate them from other marked-up documents such as HTML Web pages).

Don’t sweat it too much if you don’t include a declaration line in the XML documents you create. Leaving it out doesn’t affect how data in the document is parsed.

Elements

An *element* is a part of an XML document that contains data. If you’re accustomed to database programming or working with delimited documents, you can think of an element as a column or a field. XML elements are sometimes also referred to as *nodes*.

XML documents must have at least one top-level element to be parsable. The following code shows an XML document with a declaration and a single top-level element (but no actual data).

```
<?xml version="1.0"?>
<ORDERS>
</ORDERS>
```

This document can be parsed, even though it contains no data. Note one important thing about the markup of this document: It contains both an open tag and a close tag. The close tag is differentiated by the slash (/) character in front of the element name.

This is an important difference between XML and HTML. In HTML, some elements require close tags, but many don’t. Even for those elements that don’t contain proper closing tags, the browser often attempts to correctly render the page (sometimes with quirky results).

XML, on the other hand, is the shrewish librarian of the data universe. It’s not nearly as forgiving as HTML and will rap you on the knuckles if you cross it. If your XML document contains an element that’s missing a close tag, the document won’t parse. This is a common source of frustration among developers who use XML. Another kicker is the fact that (unlike HTML) tag names in XML are case-sensitive. This means that <ORDERS> and <orders> are considered to be two different and distinct tags.

The whole purpose of an XML element is to contain pieces of data. In the previous example, we left out the data. Code Listing 11.1 shows an evolved version of this document, this time with data in it.

LISTING 11.1 An XML Document with Elements That Contain Data

```
<?xml version="1.0"?>
<ORDERS>
  <ORDER>
    <DATETIME>1/4/2000 9:32 AM</DATETIME>
    <ID>33849</ID>
    <CUSTOMER>Steve Farben</CUSTOMER>
    <TOTALAMOUNT>3456.92</TOTALAMOUNT>
```

LISTING 11.1 Continued

```
  </ORDER>
</ORDERS>
```

If you were to describe Listing 11.1 in English, you'd say that it contains a top-level ORDERS element and a single ORDER element, or node. The ORDER node is a child of the ORDERS element. The ORDER element itself contains four child nodes of its own: DATETIME, ID, CUSTOMER, and TOTALAMOUNT.

Adding a few additional orders to this document might give you something like Listing 11.2.

LISTING 11.2 An XML Document with Multiple Child Elements Beneath the Top-Level Element

```
<?xml version="1.0"?>
<ORDERS>
  <ORDER>
    <DATETIME>1/4/2000 9:32 AM</DATETIME>
    <ID>33849</ID>
    <CUSTOMER>Steve Farben</CUSTOMER>
    <TOTALAMOUNT>3456.92</TOTALAMOUNT>
  </ORDER>
  <ORDER>
    <DATETIME>1/4/2000 9:32 AM</DATETIME>
    <ID>33856</ID>
    <CUSTOMER>Jane Colson</CUSTOMER>
    <TOTALAMOUNT>401.19</TOTALAMOUNT>
  </ORDER>
  <ORDER>
    <DATETIME>1/4/2000 9:32 AM</DATETIME>
    <ID>33872</ID>
    <CUSTOMER>United Disc, Incorporated</CUSTOMER>
    <TOTALAMOUNT>74.28</TOTALAMOUNT>
  </ORDER>
</ORDERS>
```

Here's where developers sometimes get nervous about XML. With a document like Listing 11.2, you can see that there's far more markup than data. Does this mean that all those extra bytes will squish your application's performance?

Maybe, but not necessarily. Consider an Internet application that uses XML on the server side. When this application needs to send data to the client, it first opens and parses the XML document (we'll discuss how XML parsing works later). Then some sort of result—in all likelihood, a tiny subset of the data, stripped of marku—will be sent to the client Web browser. The fact that there's a bunch of markup there doesn't slow things down significantly.

At the same time, there is a way to express data more succinctly in an XML document, without the need for as many open and closing markup tags. You can do this through the use of *attributes*.

Attributes

An attribute is another way to enclose a piece of data in an XML document. An attribute is always part of a element; it typically modifies or is related to the information in the node. In a relational database application that emits XML, it's common to see foreign key data expressed in the form of attributes.

For example, a document that contains information about a sales transaction might use attributes as shown in Listing 11.3.

LISTING 11.3 An XML Document with Elements and Attributes

```
<?xml version="1.0"?>
<ORDERS>
  <ORDER id="33849" custid="406">
    <DATETIME>1/4/2000 9:32 AM</DATETIME>
    <TOTALAMOUNT>3456.92</TOTALAMOUNT>
  </ORDER>
</ORDERS>
```

As you can see from Listing 11.3, attribute values always are enclosed in quotation marks. Using attributes tends to reduce the total number of bytes of the document, reducing some markup at the expense of readability (in some cases). Note that you are allowed to use either single or double quotation marks anywhere XML requires quotes.

This element/attribute syntax may look familiar from HTML, which uses attributes to assign values to elements the same way XML does. But remember that XML is a bit more rigid than HTML; a bracket out of place or a mismatched close tag will cause the entire document to be unparsable.

Enclosing Character Data

At the beginning of this chapter, we discussed the various dilemmas involved with delimited files. One of the problems with delimiters is the fact that if the delimiter character exists within the data, it's difficult if not impossible for a parser to know how to parse the data.

This problem is not confined to delimited files; XML has similar problems with containing delimiter characters. The problem arises because the *de facto* XML delimiter character (in actuality, the markup character) is the left angle bracket (also known as the less-than symbol). In XML, the ampersand character (&) can also throw the parser off.

You've got two ways to deal with this problem in XML: Either replace the forbidden characters with *character entities* or use a CDATA section as a way to delimit the entire data field.

Using Character Entities

You might be familiar with character entities from working with HTML. The idea is to take a character that might be interpreted as a part of markup and replace it with an escape sequence to prevent the parser from going haywire. Listing 11.4 provides an example of this.

LISTING 11.4 An XML Document with Escape Sequences

```
<?xml version="1.0"?>
<ORDERS>
  <ORDER id="33849">
    <NAME>Jones &amp; Williams Certified Public Accountants</NAME>
    <DATETIME>1/4/2000 9:32 AM</DATETIME>
    <TOTALAMOUNT>3456.92</TOTALAMOUNT>
  </ORDER>
</ORDERS>
```

Take a look at the data in the NAME element in the code example. Instead of an ampersand, the &amp; character entity is used. (If a data element contains a left bracket, it should be escaped with the &lt*;* character entity.)

When you use an XML parser to extract data with escape characters, the parser will automatically convert the escaped characters to their correct representation.

Using CDATA elements

An alternative to replacing delimiter characters is to use CDATA elements. A CDATA element tells the XML parser not to interpret or parse characters that appear in the section.

Listing 11.5 is an example of the same XML document from the previous example, delimited with a CDATA section rather than a character entity.

LISTING 11.5 An XML Document with a CDATA Section

```
<?xml version="1.0"?>
<ORDERS>
  <ORDER id="33849">
    <NAME><![CDATA[Jones & Williams Certified Public Accountants]]></NAME>
    <DATETIME>1/4/2000 9:32 AM</DATETIME>
    <TOTALAMOUNT>3456.92</TOTALAMOUNT>
  </ORDER>
</ORDERS>
```

In this example, the original data in the NAME element does not need to be changed, as in the previous example. Here, the data is wrapped with a CDATA element. The document is parsable, even though it contains an unparsable character (the ampersand).

Which technique should you use? It's really up to you. I prefer using the CDATA method because it doesn't require altering the original data, but it has the disadvantage of adding a dozen or so bytes to each element.

Abbreviated Close-Tag Syntax

For elements that contain no data, you can use an abbreviated syntax for element tags to reduce the amount of markup overhead contained in your document. Listing 11.6 demonstrates this.

LISTING 11.6 An XML Document with Empty Elements

```
<?xml version="1.0"?>
<ORDERS>
  <ORDER id="33849" custid="406">
    <DATETIME>1/4/2000 9:32 AM</DATETIME>
    <TOTALAMOUNT />
  </ORDER>
</ORDERS>
```

You can see from the example that the TOTALAMOUNT element contains no data. As a result, we can express it as <TOTALAMOUNT /> instead of <TOTALAMOUNT></TOTALAMOUNT>. (It's perfectly legal to use either syntax in your XML documents; the abbreviated syntax generally better, though, because it reduces the size of your XML document.)

Accessing XML Data

Now that you've seen how to create an XML document, we get to the fun part: how to write code to extract and manipulate data from an XML document using classes found in the .NET frameworks. There's no one right way to do this; in fact, before .NET came along, there were two predominant ways to parse an XML document: the XML DOM and Simple API for XML (SAX).

There is an implementation of the XML DOM in the .NET frameworks. In this chapter we'll primarily focus on the DocumentNavigator, XMLTextReader, and XMLTextWriter objects.

These objects are the standard .NET way to access XML data; they provide a good combination of high performance, .NET integration and ease-of-programming. But you should know about the other ways to deal with XML, too, particularly since the specialized .NET reader and writer objects are designed to interact with the Internet-standard DOM objects. So for the remainder of this chapter, we'll include brief examples of how to work with the DOM model as well.

About Simple API for XML (SAX)

Simple API for XML was designed to provide a higher level of performance and a simpler programmability model than XML DOM. It uses a fundamentally different programmability model—instead of reading in the entire document at once and exposing the elements of the document as nodes, SAX provides an event-driven model for parsing XML.

SAX is not yet supported in .NET. In fact, it's not even an official Internet standard. It's a programming interface for XML that was created by developers who wanted an XML parser with higher-performance and a smaller memory footprint, especially when parsing very large documents.

Microsoft supports an event-driven model for XML parsing known as the DocumentNavigator. This model is similar in principle to SAX, but with different implementation details. We'll cover the DocumentNavigator later in this chapter.

NOTE

Although it is not yet supported in the .NET Framework, SAX is supported in Microsoft's COM-based XML parser implementation. For more information on this tool, see `http://msdn.microsoft.com/xml/`).

Using the XML Document Object Model in .NET

The XML Document Object Model (DOM) is a programming interface used to parse XML documents. It was the first programming interface provided for XML by Microsoft; XML DOM implementations that target other languages and other operating systems are available.

The original Microsoft XML DOM implementation is COM-based, so it is accessible from any COM-compliant language. The XML parsers in .NET are, naturally, accessible from any .NET-compliant language.

The XML DOM does its magic by taking an XML document and exposing it in the form of a complex object hierarchy. This kind of hierarchy may be familiar to you if you've done client-side HTML Document Object Model programming in JavaScript or VBScript. The number of objects in XML DOM is fairly daunting; there are no less than 20 objects in the base implementation, and the Microsoft implementation adds a number of additional interfaces and proprietary extensions.

Fortunately, the number of objects you need to work with on a regular basis in the XML DOM is minimal. In fact, the XML DOM recommendation segregates the objects in the DOM into two groups, *fundamental classes* and *extended classes*. Fundamental classes are the ones that application developers will find most useful; the extended classes are primarily useful tools to developers and people who like to pummel themselves with detail.

The fundamental classes of the XML DOM as implemented in the .NET framework are XmlNode, XmlNodeList, and XmlNamedNodeMap. These classes, as well as the parent XmlDocument class, are illustrated in Figure 11.1.

FIGURE 11.1
Fundamental XML DOM objects.

Note that the XmlDocument object is technically an extended class, not a fundamental class, because it inherits from XmlNode. We're including discussion of it in this chapter because it's rather tricky to do useful stuff in XML without it. The class adds some useful file and URL-handling capabilities to XmlNode.

NOTE

The XmlNode and XmlDocument classes are found in the System.Xml namespace. The XmlDocument class inherits from System.Xml.XmlNode. A reference to the classes, properties and methods introduced in this chapter is included at the end of this chapter.

In general, to work with a XML document using the DOM, you first open the document (using the .Load() or .LoadXML() method of the XmlDocument object). The .Load() method is overloaded and can take any one of three arguments: a string, a System.IO.TextReader object, or a System.Xml.XmlReader object.

The easiest way to demonstrate how to load an XML document from a file on disk is to pass the .Load() method a string. The string can either be a local file on disk or a URL. If the string is a URL, the XmlDocument retrieves the document from a Web server. This is pretty handy—it makes you wish that every file-handling object worked this way.

Code Listing 11.7 shows an example of loading an XML document from disk using an XmlDocument object.

LISTING 11.7 Loading a Local XML File Using the XmlDocument's .Load() Method

```
<%@ Import Namespace="System.Xml" %>

<SCRIPT runat='server'>
Sub Page_Load(Sender As Object, e As EventArgs)
  Dim xd As New XmlDocument()
  xd.Load("c:\data\books.xml")
  Response.Write (xd.OuterXml)
  xd = Nothing
End Sub
</SCRIPT>
```

This code works for any XML document accessible to the local file system. Listing 11.8 demonstrates how to load an XML document that resides on a Web server.

LISTING 11.8 Loading an XML File That Resides on a Web Server

```
<%@ Import Namespace="System.Xml" %>

<SCRIPT runat='server'>
  Dim xd As New XmlDocument()
  xd.Load("http://www.myserver.com/books.xml")
  Response.Write (xd.OuterXml)
  xd = Nothing
</SCRIPT>
```

As you can see, the syntax is nearly identical whether you're loading the file from the local file system or over HTTP. Both of these examples are extremely simple; they demonstrate how easy it is to open and view an XML document using the DOM. The next step is to start doing things with the data in the document you've retrieved.

Viewing Document Data Using the XmlNode Object

Once you've loaded a document, you need some way to programmatically visit each of its nodes in order to determine what's inside. In the XML DOM, there are several ways to do this, all of which are centered around the XmlNode object.

The XmlNode object represents a node in the XML document. It exposes an object hierarchy that exposes attributes and child nodes, as well as every other part of an XML document.

When you've loaded an XML document to parse it (as we demonstrated the previous code examples), your next step will usually involve retrieving that document's top-level node. Use the .FirstChild() property to do this.

Listing 11.9 shows an example of retrieving and displaying the name of the top-level node in the document using .FirstChild().

LISTING 11.9 Loading a Local XML File Using the XmlDocument's .Load() Method

```
<%@ Import Namespace="System.Xml" %>

<SCRIPT runat='server'>
Sub Page_Load(Sender As Object, e As EventArgs)
  Dim xd As New XmlDocument()
  xd.Load("c:\data\books.xml")
  MsgBox (xd.FirstChild.Name)
  xd = Nothing
End Sub
</SCRIPT>
```

The code demonstrates how the .FirstChild() property returns a XmlNode object with its own set of properties and methods. In the example, we call the .Name() property of the XmlNode object returned by .FirstChild().

You can do more useful and interesting things with the XmlNode object. One common operation is drilling down and retrieving data from the ChildNodes object owned by XmlNode. Two features of ChildNodes make this possible: its status as an *enumerable class*, and the InnerText property of each child node.

Enumerable classes implement the .NET IEnumerable interface. This is the same interface definition that arrays, collections, and more complex constructs like ADO.NET DataSets support. (You may think of ChildNodes as just another collection, but in .NET, Collection is a distinct data type.)

When an object supports IEnumerable, it exposes functionality (through a behind-the-scenes object called an enumerator) that enables other processes to visit each of its child members. In the case of ChildNodes, the enumerator lets your code visit the object's child XmlNode objects. The For Each...Next block in Visual Basic is the construct that is most commonly used to traverse an enumerable class. Listing 11.10 shows an example of this.

LISTING 11.10 Traversing the Enumerable ChildNodes Class

```
<%@ Import Namespace="System.Xml" %>

<SCRIPT runat='server'>
Sub Page_Load(Sender As Object, e As EventArgs)
  xd.Load("c:\data\books.xml")
  ndBook = xd.FirstChild.Item("BOOK")

  For Each nd In ndBook.ChildNodes
    If nd.Name = "AUTHOR" Then
      MsgBox("The author's name is " & nd.InnerText)
    End If
  Next
End Sub
</SCRIPT>
```

In this code example, the For Each...Next loop goes through the set of XmlNode objects found in ChildNodes. When it finds one whose Name property is AUTHOR, it displays it. Note that for the example file books.xml, two message boxes will appear, because the example book has two authors.

Note also that the value contained in an XML node is returned by the InnerXml() property in .NET, not by the .text property as it was in the COM-based MSXML library. Making a more granular distinction between a simple "text" property versus inner and outer text or inner and outer XML gives you a greater degree of power and flexibility. Use the "outer" properties when you want to preserve markup; the "inner" properties return the values themselves.

With the few aspects of the XmlDocument and XmlNode objects we've discussed so far, you now have the ability to perform rudimentary retrieval of data in an XML document using the DOM. However, looping through a collection of nodes using For Each...Next leaves something to be desired. For example, what happens when your book node contains a set of 50 child nodes, and you're only interested in extracting a single child node from that?

Fortunately, .NET provides several objects that enable you to easily navigate the hierarchical structure of an XML document. These include the XmlTextReader and DocumentNavigator object.

Using the XmlDataReader Object

The XmlDataReader object provides a method of accessing XML data that is both easier to code and more efficient than using the full-blown XML DOM. At the same time, the XmlDataReader understands DOM objects in a way that lets you use both types of access cooperatively.

NOTE

XmlDataReader is found in the System.Xml namespace. It inherits from System.Xml.XmlReader, an abstract class. A reference to the classes, properties, and methods introduced in this chapter is included at the end of this chapter.

If you've used the XML DOM in the past, the XmlDataReader will change the way you think about XML parsing in general. The XmlDataReader doesn't load an entire XML document and expose its various nodes and attributes to you in the form of a large hierarchical tree; that process causes a large performance hit as data is parsed and buffered. Instead, think of the XMLDataReader object as a truck that bounces along the road from one place to another. Each time the truck moves across another interesting aspect of the landscape, you have the ability to take some kind of interesting action based on what's there.

Parsing an XML document using the XmlDataReader object involves a few steps. First, you create the object, optionally passing in a file name or URL that represents the source of XML to parse. Next, execute the .Read method of the XmlDataReader object until that method returns the value False. (You'll typically set up a loop to do this so you can move from the beginning to the end of the document.)

Each time you execute the XmlDataReader object's .Read method, the XmlDataReader object's properties are populated with fragments of information from the XML document you're parsing. This information includes the type of the data the object just read, and the value of the data itself (if any).

The type of data is exposed through the XmlDataReader object's NodeType property. The value of data retrieved can be retrieved in an untyped format (through the .Value() property of the XmlDataReader object) or typed format (through such properties as .ReadDateTime(), .ReadInt32(), .ReadString(), and so forth).

Most of the time, the NodeType property will be XmlNodeType.Element (an element tag), XmlNodeType.Text (the data contained in a tag), or XmlNodeType.Attribute.

Listing 11.11 shows an example of how this works. The objective of this example is to retrieve the title of a book from an XML file that is known to contain any one of a number of nodes pertaining to the book itself.

LISTING 11.11 Extracting a Book Title Using the XmlTextReader Object

```
<%@ Import Namespace="System.Xml" %>
<SCRIPT runat='server'>

Sub Page_Load(Sender As Object, e As EventArgs)
    Dim xr As New XmlTextReader(Server.MapPath("books.xml"))
    Dim bTitle As Boolean

    While xr.Read()
      Select Case xr.NodeType
        Case XmlNodeType.Element
          If xr.Name = "TITLE" Then
            bTitle = True
          End If

        Case XmlNodeType.Text
          If bTitle Then
            Response.Write("Book title: " & xr.ReadString)
            bTitle = False
          End If
      End Select
    End While
End Sub
</SCRIPT>
```

NOTE

This code example can be found in the downloadable code examples under the XML section in the package XmlTextReader.zip.

The example opens the XML file by passing the name of the XML file to the constructor of the XmlDataReader object. It then reads one chunk of the document at a time (through successive calls to the XmlDataReader object's Read method). If the current data represents the element name "TITLE", the code sets a flag, bTitle.

When the bTitle flag is set to True, it means "get ready, a book title is coming next." The book title itself is extracted in the next few lines of code. When the code encounters the text chunk, it extracts it from the XML document in the form of a string.

Note that the values XmlNodeType.Element and XmlNodeType.Text are predefined members of the XmlNodeType structure. You can set up more involved parsing structures based on any XML type found in the DOM if you wish. For example, if you included a case to process based on the type XmlNodeType.XmlDeclaration, you could process the XML declaration that appears (but is not required to appear) as the first line of the XML document.

As you can see from these examples, a beautiful thing about XML is the fact that if the structure of the document changes, your parsing code will still work correctly, as long the document contains a TITLE node. (In the previous code example, if for some reason the document contains no book title, no action is taken.) So the problems we discussed at the beginning of this chapter go away in the new world of XML parsing.

The XmlDataReader works well for both large and small documents. Under most circumstances (particularly for large documents), it should perform better than the XML DOM parser. However, like the DOM, it too has its own set of limitations. The XmlDataReader object doesn't have the ability to scroll—to jump around between various areas in the document. (If you're a database developer, you can think of an XmlDataReader as being analogous to a cursorless or forward-only result set.) Also, as its name implies, the XmlDataReader object only permits you to read data; you can't use it to make changes in existing node values or add new nodes to an existing document.

To provide a richer set of features, including the ability to scroll backward and forward in a document, the .NET framework provides another object, the DocumentNavigator object.

Using the DocumentNavigator Object

So far in this chapter you've seen two distinct ways provided by the .NET Framework to access XML data: the XML Document Object Model and the XmlDataReader object. Both have their advantages and drawbacks.

In many ways, the DocumentNavigator object represents the best of all worlds. It provides a simpler programmability model than the XmlDocument object, yet it integrates with the standard DOM objects nicely. In fact, in most cases when you're working with XML data in .NET, you'll typically create a DocumentNavigator by creating a DOM XmlDocument object first.

NOTE

The DocumentNavigator class is found in the System.Xml namespace. It inherits from System.Xml.XmlNavigator, an abstract class. A reference to the classes, properties, and methods introduced in this chapter is included at the end of this chapter.

Listing 11.12 shows an example of creating a DocumentNavigator object from an existing XmlDocument object that has been populated with data.

LISTING 11.12 Creating a DocumentNavigator Object from an XmlDocument Object

```
<%@ Import Namespace="System.Xml" %>
<SCRIPT runat='server'>

Sub Page_Load(Sender As Object, e As EventArgs)
  Dim xd As New XmlDocument()
  Dim xn As DocumentNavigator = New DocumentNavigator(xd)
  ' Code to work with the DocumentNavigator goes here
End Sub
</SCRIPT>
```

Navigating Through the Document Using the DocumentNavigator Object

After you've created and populated the DocumentNavigator, you can move through the document. You begin by moving to the beginning of the document by executing the DocumentNavigator's MoveToDocument method. This method is handy because it always gets you to the beginning of the document. In a way, MoveToDocument is the DocumentNavigator version of the ADO.old MoveFirst method.

NOTE

The .NET Framework SDK documentation suggests that you must execute the MoveToDocument method first, before you can begin working with a document using a DocumentNavigator. We didn't find this to be the case. Go figure.

Unfortunately, the similarities between the Recordset and the XML DocumentNavigator end there. This is because the Recordset represents a nice, tidy two-dimensional array of data; the DocumentNavigator, in contrast, provides access to XML documents that can contain complex hierarchies. So rather than starting at the top and working your way down as with a Recordset, the DocumentNavigator must provide a way for you to access subordinate child nodes of any given XML node, in addition to letting you move up and down within the node you're in currently.

The navigation methods of the DocumentNavigator object make a distinction between parent-child node relationships and sibling node relationships in an XML document. For example, in our example books.xml document, the BOOKS node is the parent of the BOOK node, and the BOOK node is the parent of the TITLE and AUTHOR nodes. You use MoveToChild to navigate between these nodes. TITLE and AUTHOR, on the other hand, are at the same level in the hierarchy; they're sibling nodes. You use MoveToNext to navigate from one sibling node to the next. Figure 11.2 illustrates this.

FIGURE 11.2
Navigating through an XML hierarchy using the DocumentNavigator object

When navigating in any given XML document, much hinges on whether the document's structure is known to you. If you can assume that a given parent node will have child nodes 100% of the time, it can save you headaches. However, before you use navigational methods such as MoveToChild, you may wish to first test to see whether children actually exist or not. You use the HasChildren method to do this.

MoveToChild is an indexed method; it takes an integer value that represents the number of the subordinate node you want to move to. (In our simple example we'll assume that each parent node only has one child node.) Because all indexes in .NET begin with zero, you pass the value zero to the MoveToChild method to move to the first child of a node.

The first time you execute MoveToChild, you're taken to the root node of the document. After that, each successive call to MoveToChild takes you deeper in the hierarchy. So for a document like books.xml that contains BOOKS, BOOK, and TITLE nodes, you'd have to execute MoveToChild three times before you landed at the first TITLE node. Listing 11.13 demonstrates this.

LISTING 11.13 Using MoveToChild to Drill down into the Hierarchy of an XML Document

```
<%@ Import Namespace="System.Xml" %>
<SCRIPT runat='server'>

Sub Page_Load(Sender As Object, e As EventArgs)
  Dim xd As New XmlDocument()
  xd.Load(Server.MapPath("books.xml"))
  Dim xn As DocumentNavigator = New DocumentNavigator(xd)

  xn.MoveToChild(0)  ' Go to BOOKS
  xn.MoveToChild(0)  ' Go to BOOK
  xn.MoveToChild(0)  ' Go to TITLE
  Response.Write(xn.Name & " - " & xn.Value & "<BR>")

End Sub
</SCRIPT>
```

Once you've moved to a node that contains data, you can move to the next sibling node using the MoveToNext method. Listing 11.14 shows an example of this, outputting all the node names and values for the book stored in the document.

LISTING 11.14 Using MoveNext to Navigate Between Sibling Nodes

```
<%@ Import Namespace="System.Xml" %>
<SCRIPT runat='server'>

Sub Page_Load(Sender As Object, e As EventArgs)
  Dim xd As New XmlDocument()
  xd.Load(Server.MapPath("books.xml"))
  Dim xn As DocumentNavigator = New DocumentNavigator(xd)

  xn.MoveToChild(0)  ' Go to BOOKS
  xn.MoveToChild(0)  ' Go to BOOK
  xn.MoveToChild(0)  ' Go to TITLE
  ' Output book title
  Response.Write("<B>" & xn.InnerText & "</B><BR>")

  Do While xn.MoveToNext()
    ' Output all authors
    Response.Write(xn.Name & " - " & xn.InnerText & "<BR>")
  Loop

End Sub
</SCRIPT>
```

This is a good demonstration of how MoveToNext serves to control the looping structure, similar to the Read method of the XmlTextReader object we discussed earlier in this chapter. Because MoveToNext returns True when it successfully navigates to a sibling node and False when there are no more nodes left to navigate to, it's easy to set up a While loop that displays data for all the nodes owned by a book.

So you've seen with the previous few examples that navigating using MoveToChild and MoveNext works well enough. But if the process of repeatedly executing the MoveToChild and MoveNext methods to drill down into the document hierarchy seems a little weak to you, you're right. For example, how do you go directly to a node when you know the name of the node and can be reasonably sure that the node exists? And how do we get rid of the inelegant process of calling MoveToChild repeatedly to drill down to the place in the document where useful data exists?

Fortunately, the DocumentNavigator provides a number of more sophisticated techniques for drilling into the document hierarchy which we'll discuss in more detail in the next few sections.

Using the Select and SelectSingle Methods to Retrieve Nodes Using an XPath Query

The Select method of the DataNavigator object enables you to filter and retrieve subsets of XML data from any XML document. You do this by constructing an XPath expression and passing it to either the Select or SelectSingle methods of the DataNavigator object. An XPath expression is a compact way of querying an XML document without going to the trouble of parsing the whole thing first. Using XPath, it's possible to retrieve very useful subsets of information from an XML document, often with only a single line of code.

NOTE

XPath syntax is described in more detail in the section "Querying XML Documents Using XPath Expressions" later in this chapter.

Listing 11.15 shows a very simple example of using an XPath expression passed to the Select method to move to and display the title of the first book in the document books.xml.

LISTING 11.15 Using the Select Method of the DocumentNavigator Object to Retrieve a Subset of Nodes

```
<%@ Import Namespace="System.Xml" %>
<SCRIPT runat='server'>

Sub Page_Load(Sender As Object, e As EventArgs)
  Dim xd As New XmlDocument()
  xd.Load(Server.MapPath("books.xml"))
  Dim xn As DocumentNavigator = New DocumentNavigator(xd)

  xn.MoveToDocument()
  xn.Select("BOOKS/BOOK/AUTHOR")
  xn.MoveToNextSelected()
  Response.Write(xn.Name & " - " & xn.InnerText)

End Sub
</SCRIPT>
```

When the Select method in this example is executed, you're telling the DocumentNavigator object to retrieve all of the AUTHOR nodes owned by BOOK nodes contained in the BOOKS root node. The XPath expression "BOOKS/BOOK/AUTHOR" means "all the authors owned by BOOK nodes under the BOOKS root node." Any AUTHOR nodes in the document owned by parent nodes other than BOOK won't be retrieved, although you could construct an XPath expression to retrieve AUTHOR nodes anywhere in the document regardless of their parentage.

The product of this operation is a *selection*, a subset of XML nodes that can then be manipulated independently of the main document. After you've retrieved a selection, you then execute the MoveToNextSelected method to move to the first selected node. From there you can retrieve and display the data from the selected nodes (potentially calling MoveToNextSelected again to loop through all the selected nodes).

This example is useful, but it has a flaw: It only displays the first author, and this book has two authors! In this case, the Select method did indeed retrieve all the AUTHOR nodes owned by the BOOK node; we just didn't display them. To display all of them, we need to create a loop. Listing 11.16 demonstrates how to do this.

LISTING 11.16 Using the Select Method of the DocumentNavigator Object to Display All Book Authors

```
<%@ Import Namespace="System.Xml" %>
<SCRIPT runat='server'>

Sub Page_Load(Sender As Object, e As EventArgs)
  Dim xd As New XmlDocument()
  xd.Load(Server.MapPath("books.xml"))
  Dim xn As DocumentNavigator = New DocumentNavigator(xd)

  xn.MoveToDocument()
  xn.Select("BOOKS/BOOK/AUTHOR")

  While xn.MoveToNextSelected()
    Response.Write(xn.Name & " - " & xn.InnerText & "<BR>")
  End While

End Sub
</SCRIPT>
```

In this example, we're taking advantage of the fact that the MoveToNextSelected method returns a Boolean True/False value based on whether there are any more nodes in the selection to retrieve. If there is no next node, the method returns False and your loop exits. This code will work for zero, one, or many authors in a document.

The DataNavigator object also gives you a way to explicitly display the first match returned by an XPath query. The SelectSingle method retrieves a single node that matches the XPath expression. Be careful when using SelectSingle, though. You'll want to ensure that the document doesn't have more than one instance of the node you're looking for, because the method will only select the first node that matches your expression.

Querying XML Documents Using XPath Expressions

XPath is a set-based query syntax for extracting data from an XML document. If you're accustomed to database programming using Structured Query Language (SQL), you can think of XPath as being somewhat equivalent to SQL. But as with so many analogies between relational and XML data, the similarities run out quickly. XPath demands a completely different implementation to handle the processing of hierarchical data.

NOTE

The XPath syntax is a World Wide Web Consortium (W3C) recommendation. You can get more information about XPath from the W3C site at `http://www.w3.org/TR/xpath`. Information on the Microsoft XML 3.0 (COM) implementation of XPath is at `http://msdn.microsoft.com/library/psdk/xmlsdk/xslr0fjs.htm`.

The idea behind XPath is that you should be able to extract data from an XML document using a compact expression, ideally on a single line of code. Using XPath is generally a more concise way to extract information buried deep within an XML document. (The alternative to using XPath is to write loops or recursive functions, as most of the examples used earlier in this chapter did.) The compactness of XPath comes at a price, though: readability. Unless you're well versed in the XPath syntax, you may have trouble figuring out what the author of a complicated XPath expression was trying to look up. Bear this in mind as you utilize XPath in your applications.

While the complete XPath syntax is quite involved (and beyond the scope of this book), there are certain commonly used operations you should know about as you approach XML processing using the .NET framework classes. The three most common XPath scenarios include:

- Retrieving a subset of nodes that match a certain value (for example, all of the orders associated with customers)
- Retrieving one or more nodes based on the value of an attribute (such as retrieving all of the orders for customer ID 1006)
- Retrieving all the parent and child nodes where an attribute of a child node matches a certain value (such as retrieving all the customers and orders where the Item attribute of the order node equals "Tricycle")

Creating XSD Schemas

Whenever you utilize or manipulate data, you need to have a way of answering certain questions about that data. Do you define an Invoice ID as a textual or numeric value? Is a phone number limited to ten digits? More?

There are several ways to do this. First, you can simply provide validation logic in code, just as any software application would. This defeats the purpose of XML on a number of levels, though. Remember that XML is designed to be interoperable and human-readable. When you commit validation logic to code, you've almost inherently made the validation logic inaccessible to other processes that might come along later.

Document Type Definitions (DTDs)

The first technology used for defining XML structures was known as the Document Type Definition (DTD). The problem with DTDs is that they have their own weird syntax that has nothing to do with XML. A good example of a DTD is the DTD for XML itself, which resides at `http://www.w3.org/XML/1998/06/xmlspec-v21.dtd`.

Microsoft chose to use a more evolved document definition technology for XML in the .NET universe—the XML schema. Visual Studio.NET gives developers a graphical way to build XML schemas and contains little or no support for DTDs. For this reason, this book will focus on schemas rather than DTDs.

Class Reference

This section provides a reference to the key objects described in this chapter.

Inheritance Relationships

This chapter covers three classes involved in XML handling in the .NET framework. The XmlDocument object provides access to the XML DOM through .NET. The XmlTextReader and DocumentNavigator classes are a .NET-specific way to handle XML documents.

XmlTextReader and DocumentNavigator both inherit from abstract classes found in the System.Xml namespace. Figure 11.3 shows the inheritance relationship between these objects.

FIGURE 11.3
Inheritance relationships between proprietary .NET XML handler classes

NOTE

Because the DocumentNavigator class inherits from the abstract XmlNavigator class, it implies that there could be other subclassed implementations of XmlNavigator. In fact, the .NET framework contains just such a class, the DataDocumentNavigator, discussed in Chapter 12.

To make this reference more concise, we omit descriptions for inherited properties, methods and events that are not significantly different from those found in base classses. In the following listings, such members are displayed in italic type.

XmlDocument Object

This object is a member of System.Xml.

The XmlDocument object represents the top-level object in the XML DOM hierarchy. Properties and methods listed in italics are not described in this section, typically because they are inherited or otherwise duplicated from base classes described elsewhere.

Constructor Examples

```
xd = New System.Xml.XmlDocument()
xd = New System.Xml.XmlDocument(XmlNameTable)
```

Properties

Attributes	IsDocumentReadOnly	NodeType
ChildNodes	IsReadOnly	OuterXml
DocumentElement	Item	OwnerDocument
DocumentType	LastChild	ParentNode
FirstChild	LocalName	Prefix
HasChildNodes	Name	PreserveWhitespace
Implementation	NamespaceURI	PreviousSibling
InnerText	NameTable	Value
InnerXml	NextSibling	

Methods

AppendChild	CreateWhitespace	Load
Clone	CreateXmlDeclaration	LoadXml
CloneNode	*Equals*	*MemberwiseClone*

CreateAttribute	*Finalize*	Normalize
CreateCDataSection	GetElementById	PrependChild
CreateComment	GetElementsByTagName	ReadNode
CreateDocumentFragment	GetEnumerator	RemoveAll
CreateDocumentType	*GetHashCode*	RemoveChild
CreateElement	GetNamespaceOfPrefix	ReplaceChild
CreateEntityReference	GetPrefixOfNamespace	Save
CreateNode	*GetType*	Supports
CreateProcessingInstruction	ImportNode	*ToString*
CreateSignificantWhitespace	InsertAfter	WriteContentTo
CreateTextNode	InsertBefore	WriteTo

AppendChild Method

The AppendChild method adds a new node to the end of the XML document. It is inherited from System.Xml.XmlNode. The method takes an XmlNode object as its sole argument and returns an XmlNode object.

Note that in general, unless you're using AppendChild to construct an XML document from scratch, you don't want to use the AppendChild method against the document itself. (This is an illegal operation, because an XML document can only contain a single root node.) To add nodes to an existing document, use the AppendChild method of the document's root node (accessible through its FirstChild property).

An example is shown in the following code.

```
Dim xd As New XmlDocument()
Dim xd As XmlNode
xd.LoadXml("<XML>My Document</XML>")
nd = xd.CreateNode(XmlNodeType.Element, "NEWNODE", "")
nd.InnerText = "New Data"
xd.FirstChild.AppendChild(nd)
```

Attributes Property

The Attributes property returns an XmlAttributesCollection that contains all the elements in the document.

DocumentNavigator Object

This object is a member of System.Xml

The DocumentNavigator enables you to navigate an XML document using a scrolling cursor model. It inherits from System.Xml.XmlNavigator, an abstract class.

Constructors

```
dn = New DocumentNavigator(xd As XmlDocument)
dn = New DocumentNavigator(st As System.Xml.DocumentNavigator.NavState)
```

Properties

AttributeCount	*InnerText*	NamespaceURI
ChildCount	*InnerXml*	NameTable
HasAttributes	IsDefault	NodeType
HasChildren	IsEmptyTag	OuterXml
HasSelection	IsReadOnly	Prefix
HasValue	LocalName	Selection
IndexInParent	Name	Value

Methods

Clone	Move	MoveToNext
Compile	MoveChildren	MoveToNextAttribute
CopyChildren	MoveSelected	MoveToNextSelected
CopySelected	MoveTo	MoveToParent
Equals	MoveToAttribute	MoveToPrevious
Evaluate	MoveToChild	MoveToPreviousSelected
Finalize	MoveToDocument	PopPosition
GetAttribute	MoveToDocumentElement	PushPosition
GetHashCode	MoveToElement	Remove
GetNode	MoveToFirst	RemoveChildren
GetType	MoveToFirstAttribute	RemoveSelected
HasAttribute	MoveToFirstChild	Select
Insert	MoveToFirstSelected	SelectSingle
IsSamePosition	MoveToId	SetAttribute
LookupPrefix	MoveToLast	*ToString*
Matches	MoveToLastChild	
MemberwiseClone	MoveToLastSelected	

AttributeCount Property

The AttributeCount Property is an integer that represents the number of attributes associated with the current node. Note that the value of this property is predicated on the current node and will change as you traverse the document.