

# 5

## ADO.NET Integration with XML

*Extensible Markup Language*, usually referred to as XML, is a very simple and flexible format for defining data and data structures. It has been an important tool for developers for years and is rapidly becoming the standard for exchanging data between applications and platforms due to its extreme flexibility and the ease with which it can be consumed by any operating system.

What makes XML so flexible is the fact that it's a metalanguage. Unlike ridged languages such as HTML that follow a defined format, metalanguages describe the format of another language. By providing a language to describe the format of the data, XML enables you to create an infinite number of types of XML documents to store almost any type of data you may have.

In addition to the extremely flexible format, XML has another huge advantage over other data storage formats: it is not application- or operating-system-specific. XML is typically stored as seven-character ASCII text that can be interpreted by any platform or application. Compared to other formats, it is easily readable by humans, and readers usually find that what the data elements represent is intuitive.

All of these advantages have led to XML being widely integrated into the .NET Framework. As a .NET developer, you see it everywhere. Sometimes you are explicitly working with it, such as when working with configuration files or performing XSL transformations. Other times, it is being used by the .NET Framework almost invisibly to you, such as when working with a dataset, or when writing and calling XML Web Services. Either way, knowing how to work with XML and how XML integrates with the .NET Framework are important skills and essential building blocks for understanding how ADO.NET works.

## Chapter 5

### What This Chapter Covers

This chapter provides a brief overview of what XML features were available in the 1.x Framework. You will learn about enhancements to the `XmlReader` and `XmlWriter` objects that help simplify and consolidate much of the functionality in the 1.0 Framework. You will also examine some of the designer enhancements that help provide a better user experience, and then learn about the `XPathDocument`, which has evolved into a more feature-rich object for working with and editing XML documents. Finally, you will learn about the performance gains with the 2.0 Framework and what new features you can expect to see in future releases of the .NET Framework.

You should have previous experience with XML before reading this chapter. In addition, in order to run the examples provided, you will need a copy of Microsoft Visual Studio 2005. Most of the examples build from the XML file and XSD schema that follow, which are also available for download from this book's Web site at [www.wrox.com](http://www.wrox.com). The XML file is an XML representation of a few records from the pubs sample database included with Microsoft SQL Server:

```
<?xml version="1.0" encoding="UTF-8"?>
<pubs>
  <titles name="The Busy Executive's Database Guide" pub_id="1389"
price="19.99">
    <authors au_lname="Green" au_fname="Marjorie"/>
    <authors au_lname="Bennet" au_fname="Abraham"/>
  </titles>
  <titles name="Cooking with Computers: Surreptitious Balance Sheets" pub_id="1389"
price="11.95">
    <authors au_lname="O'Leary" au_fname="Michael"/>
    <authors au_lname="MacFeather" au_fname="Stearns"/>
  </titles>
  <titles name="You Can Combat Computer Stress!" pub_id="0736" price="2.99">
    <authors au_lname="Green" au_fname="Marjorie"/>
  </titles>
  <publishers pub_id="0736" pub_name="New Moon Books"/>
  <publishers pub_id="0877" pub_name="Binnet & Hardley"/>
  <publishers pub_id="1389" pub_name="Algodata Infosystems"/>
</pubs>

<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="pubs">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="titles" maxOccurs="unbounded"/>
      <xsd:element ref="publishers" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="titles">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="authors" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

```
<xsd:attribute name="pub_id" type="xsd:integer"/>
<xsd:attribute name="price" type="xsd:float"/>
</xsd:complexType>
</xsd:element>
<xsd:element name="publishers">
  <xsd:complexType>
    <xsd:attribute name="pub_id" type="xsd:integer"/>
    <xsd:attribute name="pub_name" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>
<xsd:element name="authors">
  <xsd:complexType>
    <xsd:attribute name="au_lname" type="xsd:string"/>
    <xsd:attribute name="au_fname" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

## Where XML Is Today

When XML was first introduced to the majority of developers not too many years ago, it was promoted as the solution to every data storage and application interoperability problem. So far, it has fallen short of this universal solution, but it has still proven itself very useful.

### Data Exchange

Since XML was introduced, it has been applied to a wide variety of purposes. Probably the most significant of these is a format for exchanging data between applications. Before XML, the standard means of exchanging disconnected data was either a flat text file or some form of delimited text file such as CSV. Often it was much worse, such as a propriety binary data format. Over the last few years, the standard has been shifted to using XML. This greatly simplifies the process of exchanging data between apps by providing a single standard that is easy to parse and easily human readable. It also provides a means to easily convert the data from the format of one application to that of another by using stylesheets.

### XML Web Services

Providing a standard format for exchanging data was a huge step forward, but that is only half of the equation. The other half is providing a standard method for easily transmitting this data between applications. XML Web Services have expanded on these features to provide an easy way for applications to connect to one another and transfer data. This was very difficult to do before when the calling and receiving applications were not written in the same programming language or running on the same operating system. Because XML Web Services are typically run on top of a Web server using HTTP or HTTPS, it is very easy for businesses to migrate to XML Web Services. They are now the standard means for transmitting data across the Internet in real time.

The .NET Framework has abstracted away the details of calling XML Web Services to the point that you don't really even need to know XML to use them. It has done this by automatically generating the proxy classes that handle all the work. To call a Web service, you simply add a Web reference and refer to it in

## Chapter 5

---

code as if you were calling a local DLL. To write a Web service, you just add a few attributes to your class and methods. This ease of use has made XML Web Services particularly appealing to .NET developers.

### Configuration Files

Another area in which XML has been widely implemented is configuration files. In Windows 3.x, most configuration settings were stored in flat configuration files. With the release of Windows 95, Microsoft moved to storing all Windows and application settings in the system registry, enabling it to function as a single repository that could be accessed via common APIs. Unfortunately, as the registry's size grows, the performance degrades; and if the registry became corrupt, it was very difficult to recover. With the introduction of the .NET Framework, Microsoft moved to using XML files as the primary store for configuration settings. This combined the benefits of having an easily deployable, human readable text file along with the capability to use common APIs to work with the data. It also cleaned up some security holes by no longer requiring the application that needs to read the configuration files to have access to the registry.

.NET developers have followed this model, storing their configuration settings as XML. This has been accomplished in one of two ways. The first method stores them in default configuration files that are automatically inherited by the application, such as `app.config`, `machine.config`, or `web.config`. The second method stores them in separate configuration files that are then read by the application. With either approach, customizing the application settings is much easier than previous methods.

### Text Markup

XML is also commonly used to mark up text designed for display to make it easier to change the display format. One example common to .NET developers is the use of XML comments, which enable developers to document their code in a structured format using XML. This XML is extracted during compilation to produce an XML output file. This output file can be parsed via code or transformed using XSLT to output the documentation in any format desired. A common tool for doing this is NDOC, an open-source utility for producing MSDN and other styles of documentation from the XML file.

XML has also become very popular with the recent increase of Web logs, or *blogs* for short. Bloggers write log entries that are stored as XML and can be easily presented on a Web site as HTML or consumed and consolidated with other Web logs and presented in pretty much any format desired.

The preceding examples are just a few of the more popular uses of XML today. There are far too many uses to list them all, and more are being invented every day. You can see how quickly it has gained popularity over the last few years and how important it is to understand how to work with it and where it can be used.

## Design Goals for System.Xml 2.0

A wide variety of changes have been made to `System.Xml` in the 2.0 release of the .NET Framework. Most of these enhancements revolve around just a few design goals, which you'll look at more closely throughout the chapter:

- Improved performance
- Improved schema support
- Enhanced security
- Better usability

## XmlReader and XmlWriter

The `XmlReader` and `XmlWriter` classes introduced in the 1.0 version of the .NET Framework are extremely valuable tools to most XML developers. They allow very quick, forward-only reading and writing of XML files. In the 2.0 Framework, Microsoft has introduced some new features for these classes that make working with them even easier.

### Factory Methods

Some new static creation methods are available for both the `XmlReader` and `XmlWriter` classes. These methods are designed for a number of purposes. One of these is to simplify development by not requiring the developer to know which `XmlReader` and `XmlWriter` to use.

Currently, if you want to simply read an XML file with no special options, you don't create an `XmlReader`; you create an `XmlTextReader`. Similarly, you may create an `XmlNodeReader` or `XmlValidatingReader`. Each of these `XmlReaders` are optimized to perform their specific tasks. If the same course is followed as new optimizations are added, so are more classes, which overcomplicates working with `XmlReaders`. The same is true for the `XmlWriters`. The static create methods have greatly simplified this. To read an XML file from one source and write it to another, your code will now be as simple as this:

```
Dim reader As XmlReader
Dim writer As XmlWriter

reader = XmlReader.Create("pubs.xml")
writer = XmlWriter.Create("output.xml")

While reader.Read()
    writer.WriteNode(reader, True)
End While

reader.Close()
writer.Close()
```

The preceding example is great for replacing the `XmlTextReader`, but it's not really optimized for other tasks such as validation. The `create` methods are overloaded in order to handle these optimizations. There is an `XmlReaderSettings` class for setting the options for the `XmlReader`, and an `XmlWriterSettings` class for doing the same with the `XmlWriter`. Both of these have several properties you can set, ranging from validating the document or filtering out nodes, to simply formatting the document the way you want it to appear. Following is an example of some of the tasks you can perform by using these settings classes:

## Chapter 5

---

```
Dim reader As XmlReader
Dim writer As XmlWriter
Dim readerSettings As New XmlReaderSettings()
Dim writerSettings As New XmlWriterSettings()

readerSettings.IgnoreComments = True
readerSettings.Schemas.Add(Nothing, "pubs.xsd")
readerSettings.ValidationType = ValidationType.Schema

writerSettings.OmitXmlDeclaration = True
writerSettings.Indent = True
writerSettings.NewLineOnAttributes = True

reader = XmlReader.Create("pubs.xml", readerSettings)
writer = XmlWriter.Create("output.xml", writerSettings)

While reader.Read()
    writer.WriteNode(reader, True)
End While

reader.Close()
writer.Close()
```

This example tells the reader not to process any comments and performs schema validation against the `pubs.xsd` file by setting the corresponding properties. Similarly, it tells the writer not to write the XML declaration line, to indent all of the XML elements, and to add line breaks for each attribute. Doing this makes the XML output easier to read by humans, but increases the output size. This is just an example of what can be done. You can use these and the other settings in any combination you desire to get the precise output you want.

One of these new settings that is particularly worthwhile is `ConformanceLevel`. In version 1.0 of the .NET Framework, the `XmlReader` and `XmlWriter` were not conformant to the XML 1.0 standard by default. With the 2.0 Framework, you can choose the conformance level by setting the `ConformanceLevel` property. You can set this option to document or fragment conformance, or choose `auto` to have it auto-detect depending on the nodes encountered.

### **Conversion Between XML Types and Framework Types**

Converting between XML schema types and .NET Framework types is a fairly routine task that wasn't as simple as it could be in the 1.0 Framework. Before, it was necessary to use the `XmlValidatingReader` and `XmlConvert` in order to perform the conversion. The 2.0 Framework simplifies this task with the introduction of several new `ReadContentAs...` methods. For example, to return the price of a book, you could use `ReadContentAsDouble` to return the value without having to later convert it to a double. The following example shows how you could calculate the total price of all of the books from the XML example provided at the beginning of the chapter:

```
Dim reader As XmlReader
Dim totalPrice As Double = 0
reader = XmlReader.Create("pubs.xml")
While reader.Read()
    If reader.IsStartElement() = True And reader.Name = "titles" Then
        reader.MoveToAttribute("price")
        totalPrice += reader.ReadContentAsDouble()
    End If
End While
```

Of course, the same is true when using the `XmlWriter`. The Framework now has the capability to convert between CLR data types and XML schema types. The following example shows how you can programmatically write out an XML document containing attributes of types string and double using the new `WriteValue()` method:

```
Dim writer As XmlWriter
writer = XmlWriter.Create("output.xml")
writer.WriteStartDocument()
writer.WriteStartElement("pubs")
writer.WriteStartElement("titles")
writer.WriteStartAttribute("name")
writer.WriteValue("The Busy Executive's Database Guide")
writer.WriteEndAttribute()
writer.WriteStartAttribute("price")
writer.WriteValue(19.99)
writer.WriteEndAttribute()
writer.WriteEndElement()
writer.WriteEndElement()
writer.Close()
```

## Other `XmlReader` Enhancements

In addition to the items already mentioned, there are several other enhancements to the `XmlReader`. Most of these aren't revolutionary, but they will save you some time and are worth mentioning:

- ❑ **ReadSubTree**— This method will return a new `XmlReader` instance containing the current node and all of its child nodes. You can then call the `Read` method in it to loop through each child node. Once the new `XmlReader` has been closed, the original `XmlReader` will advance to the next node past the results of the subtree.
- ❑ **ReadToDescendant**— This method advances the current `XmlReader` to the descendent node with the specified name if a match is found. It will also return a Boolean indicating whether the match was found. This provides a much easier way of quickly getting to a specific node.
- ❑ **ReadToNextSibling**— This method provides an easy means for skipping over all of the child nodes to access the next sibling node with the specified name.

## Chapter 5

# Designer Enhancements

Several enhancements to the XML designer make it easier to use. These range from simple coloring of the elements, attributes, and values to fully integrated XSL debugging. In this section, you will learn how to use these features to work more efficiently.

## XML Designer

Figure 5-1 shows many of the features that make working with the XML designer easy. Please note that the color features mentioned do not appear in this black-and-white figure.

- XML nodes are now collapsible and expandable, much like .NET code or how Internet Explorer renders XML.
- The open and close tags of the node being edited are bolded, as shown in the last `titles` node in the figure.
- Any lines changed since the last time the XML document was saved are easily identifiable by the yellow highlighting to the left of the line. This is indicated on the last `titles` node shown in the figure.
- Any lines saved since the XML document was opened in the editor are now highlighted in green. See the first `publishers` node shown in Figure 5-1.

Figure 5-1

## ADO.NET Integration with XML

- ❑ You can easily override the schema and stylesheets you wish to use for debugging without modifying the actual XML document by setting the values in the document properties window.
- ❑ When you have specified a schema document to validate against, you will receive intellisense indicating which elements and attributes are available.
- ❑ Real-time well-formedness checks are available, which indicate any errors with red squiggles and an error in the error list.
- ❑ Real-time XSD validation is offered, indicating errors with blue squiggles and a warning in the error list.
- ❑ You now have the capability to quickly jump between start and end XML tags by using CTRL+].
- ❑ There is a go to Definition option when right-clicking on a node for quickly hopping to the XSD Schema.
- ❑ You can easily preview the XSL transformations by using the Show XSL Output option from the XML menu.

### **XSL Debugging**

Because an XSL document is also an XML document, you have all of the features mentioned previously when working with XML documents. You also have an extra feature that is very helpful: the capability to debug XSL transformations.

To begin debugging an XSL document, open it in the designer. Then open the properties window and specify the values for the input document and output document. After doing so, set a breakpoint in the XSL document the same way you would in code. Notice in Figure 5-2 that only the `xs1:value` element is highlighted. This is because the breakpoints are set at the node level, not the line level.

Once the breakpoint is set, click the Debug XSL option from the XML menu or click the run button from the XSL toolbar to begin debugging. The debugger will begin running the transformation, showing the output in a new window opened to the right of the XSL document. You then have the capability to step into your transformation just as you would with code and see your XSL document as it is forming.

## Chapter 5

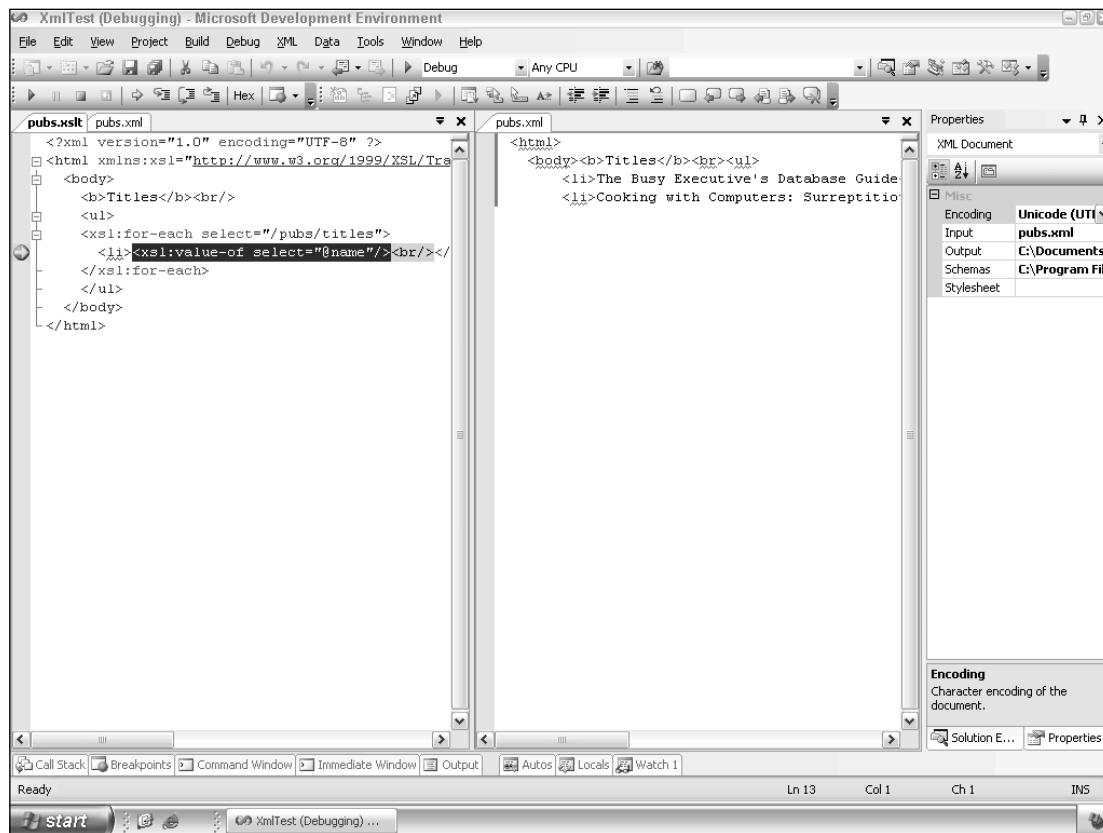


Figure 5-2

## XSD Enhancements

Just as with XSL, an XSD document is also an XML document, so the designer for XSD documents supports all of the same features as the XML designer. Other than this, there aren't any enhancements to the XSD designer, but several features make working with XSD documents easier. Most of these revolve around validating the XML document against a schema and were mentioned previously in the section on XML designer features.

There is one feature we haven't covered yet, however. You no longer need to manually write the XSD document. If you already have an XML document but not an XSD, Visual Studio will generate one for you. To do this, load the XML document in the designer and from the XML menu, choose Create Schema. This will generate the XSD schema based upon the patterns detected in the XML document and automatically associate the XML document with the newly generated XSD. It may still be necessary to open the XSD schema and change or add some of the restrictions, but it is definitely a major time saver. Already have a DTD or XDR schema associated with the document? No problem; it will convert it to XSD for you.

## Security

Security was not a major focus of the 1.0 release of `System.Xml`. As a result, a few vulnerabilities exist in the 1.0 Framework. These have been corrected in the 2.0 release.

### Denial of Service Attacks

In the 1.0 Framework, it is possible to launch a denial of service attack with DTDs using a method known as *internal entity expansion*. This basically refers to performing a recursive definition of an entity, as in the following example:

```
<!DOCTYPE myEntity [  
  <!ENTITY hw0 "Hello World">  
  <!ENTITY hw1 "&hw0;&hw0;">  
  <!ENTITY hw2 "&hw1;&hw1;">  
  <!ENTITY hw3 "&hw2;&hw2;">  
  ...  
  <!ENTITY hw99 "&hw98;&hw98;">  
  <!ENTITY hw100 "&hw99;&hw99;">  
>  
<myEntity>&hw100;</myEntity>
```

This will cause the words "Hello World" to be repeated  $2^{100}$  (1,267,650,600,228,229,401,496,703,205,376) times, either causing extreme memory usage or completely taking down the server. Nonetheless, this is a completely legitimate DTD instruction, and any attempt to close this vulnerability would also likely eliminate needed DTD capabilities. To compensate for this, the `XmlReaderSettings` class now contains a `ProhibitDTD` property for disabling DTD parsing when it is not needed. This fix was released as a patch for the 1.1 Framework in July 2004, but is now integrated with the 2.0 Framework.

### Code Access Security

One of the components that makes the .NET Framework secure is *Code Access Security (CAS)*. It enables the machine administrator to set policies that define how much access a block of code has to the system, based upon the source of that code.

CAS works by gathering evidence about the assembly and assigning it to a code group based upon that evidence. Some examples of this evidence would be where the source document was loaded from — the local machine, a network share, a URL, and so on. If it was loaded from a URL, another factor would be that URL. By default, the code group to which an assembly is assigned is based on its security zone: Local Machine, Local Intranet, Internet, Restricted, or Trusted. You can also define your own code groups.

Each code group is defined by a single membership condition. The zones listed previously are an example of a membership condition, but additional ones include the following:

- All Code
- Application
- Application Directory
- Domain Application
- GAC (Global Assembly Cache)

## Chapter 5

---

- Hash
- Publisher
- Site
- String Name
- URL

You can also define custom membership conditions. The final step is to define the permission set that the code group has. This can range widely, from writing to the hard drive, printing, accessing the registry, accessing the Web, and so on. You can configure these settings through the Framework Configuration MMC by loading %Systemroot%\Microsoft.NET\Framework\v2.0.XXXXX\Mscorcfg.msc.

All of these same features available for code security are now available for XML security. For example, when you load an XML document using the `XmlReader`, an `Evidence` property is populated with all of the relevant information about where the document came from. The document is then prevented from performing any actions that are not defined by the code group to which it is assigned, such as loading malicious URLs. This evidence is passed up the chain of any classes using the `XmlReader`, ensuring code access security throughout the XML classes.

## XPathDocument

The `XPathDocument` is not new in version 2.0 of the Framework. It was the preferred data store for XML data that was to be used to perform XPath queries or XSL transformations. There are several new features to the `XPathDocument` in the 2.0 Framework, however, and this section covers some of the more dramatic changes.

### Editing

The reason why the `XPathDocument` was the preferred data store for XPath queries and XSL transformations in the 1.0 Framework is because it offered a significant performance gain over the `XmlDocument` class. However, it was often necessary to edit the XML document in some way before performing the transformation. When this situation arose, the best approach was typically to create an `XmlDocument`, make the changes, and use it as the store for the query. By doing this, you would lose the performance benefits associated with using the `XPathDocument`. This is no longer a problem.

What makes the `XPathDocument` so much faster is that it does not have the XML 1.0 serialization constraints, so it can treat the document as just data. Now the `XPathNavigator` API, has been enhanced to extend cursor-style editing capabilities to the `XPathDocument`. It does this by reflecting the data within the `XPathDocument` to manipulate the XML. With this enhancement, the `XPathDocument` can continue to not be bound by these constraints, and deliver high performance while still being fully editable.

This is a different approach from before and it takes some getting used to, but once you see a few examples it should start to make sense. Let's start by looking at a very simple example. Let's rename the publisher "New Moon Books" to "Full Moon Books". Start as usual by creating a new `XmlDocument` and loading the `pubs.xml` file we've been using for all of the examples. Call the `CreateNavigator` method to get an `XPathNavigator` for the document and then use the `SelectSingleNode` method of this navigator to select the publisher with `pub_id` of 0736. This query will return another `XPathNavigator`

## ADO.NET Integration with XML

object. You can now use the `SetValue` method of this navigator to change the value to “Full Moon Books”. Finally, call the `Save` function on the original `XPathDocument` to save the results to the output file. Your code should look something like this:

```
Dim doc As System.Xml.XmlDocument
Dim navigator As System.Xml.XPath.XPathNavigator

doc = New System.Xml.XmlDocument()
doc.Load("pubs.xml")

navigator = doc.CreateNavigator.SelectSingleNode _
    ("/pubs/publishers[@pub_id='0736']/@pub_name")

navigator.SetValue("Full Moon Books")
doc.Save("output.xml")
```

Run the code and you’ll notice that the publisher’s name has been replaced. It was possible to do the same thing with the `XmlDocument` class in the 1.0 version of the Framework, but you didn’t get the same performance benefits. Moreover, the result in this case would have been an `XmlNode` or `XmlNodeList` if you used the `SelectNodes` method, which does not offer all of the functions that the `XPathNavigator` does, such as the capability to run another query on the results.

Usually, the changes that need to be made to the XML document are not as simple as swapping out a value for a single property. For example, you may need to add a new publisher altogether. The `XPathNavigator` class makes this easy to do by exposing the `AppendChild` method. This method returns an instance of an `XmlWriter` object for writing new nodes. This is very convenient because most developers should already be familiar with the `XmlWriter`, eliminating the need to learn a second method of writing nodes.

Let’s give it a try. Start out as you did in the previous example by creating a new `XmlDocument` and loading the `pubs.xml` file. Again create an `XPathNavigator` by calling the `SelectSingleNode` method, but this time just select the “/pubs” node. Now call the `AppendChild` method of the navigator object to create an `XmlWriter` instance. With the writer, call the `WriteStartElement` method to create a new `publishers` element. Then call the `WriteAttributeString` method to specify the `pub_id` and `pub_name` attributes, and the `WriteEndElement` method to close the `publishers` element. Don’t forget to call the `Close` method of the writer to push the changes back to the navigator object. Finally, once again call the `Save` method of the document to save the changes to the output file:

```
Dim doc As System.Xml.XmlDocument
Dim navigator As System.Xml.XPath.XPathNavigator
Dim writer As System.Xml.XmlWriter

doc = New System.Xml.XmlDocument()
doc.Load("pubs.xml")
navigator = doc.CreateNavigator.SelectSingleNode("/pubs")
writer = navigator.AppendChild()
writer.WriteStartElement("publishers")
writer.WriteAttributeString("pub_id", "1234")
writer.WriteAttributeString("pub_name", "Wrox Press")
writer.WriteEndElement()
writer.Close()
doc.Save("output.xml")
```

## Chapter 5

If you view the output, you should see the new publisher as the last child of the `pubs` element. If you don't want the new node to be added as the last child, alternatives are available to `AppendChild`, including `InsertBefore`, `InsertAfter`, and `PrependChild`.

Now that you've learned how to replace a single value or insert a single node, let's take a look at how to do a bulk change to a document. The `Select` method of the `XPathDocument` returns an `XPathNodeIterator` that contains a collection of `XPathNavigator` objects that you can use to run sub-queries. This sounds more complicated than it is. Basically, it's as simple as the `for each` statement in the following example, which returns an `XPathNavigator` for each result of the original query. With this collection of `XPathNavigators`, you can easily loop through the results and make changes throughout the entire document.

That is exactly what the following example demonstrates. It shows how to find all of the titles written by Marjorie Green, remove any co-authors, and add a new co-author of Stearns MacFeather to each title. Begin as you did for the other two examples by creating an `XmlDocument` and loading the `pubs.xml` file. Then call the `Select` method to execute a query that returns all of the titles written by Marjorie Green. Wrap a `for each` statement around that call to return an `XPathNavigator` for each result. With this editor, call the `SelectSingleNode` method to find a co-author that isn't Green (if one exists), which will return a new `XPathNavigator`. Ensure that you received a result back by checking whether the new editor is null. If so, call the `DeleteSelf` method of that navigator to remove the co-author. Now all that is left to do is to add the co-author of Stearns MacFeather to each title. The original editor is still pointing at the title node, so call the `AppendChild` as you did in the previous example to create a new authors node. Set the attributes, close the writer, and save the document. Your code should look similar to this:

```
Dim doc As System.Xml.XmlDocument
Dim navigator, navigator2 As System.Xml.XPath.XPathNavigator
Dim writer As System.Xml.XmlWriter

doc = New System.Xml.XmlDocument
doc.Load("pubs.xml")

For Each navigator In _
    doc.CreateNavigator.Select("/pubs/titles[authors/@au_lname='Green']")

    navigator2 = navigator.SelectSingleNode("authors[@au_lname!='Green']")
    If Not IsNothing(navigator2) Then
        navigator2.DeleteSelf()
    End If

    writer = editor.AppendChild()
    writer.WriteStartElement("authors")
    writer.WriteAttributeString("au_lname", "MacFeather")
    writer.WriteAttributeString("au_fname", "Stearns")
    writer.Close()
Next

doc.Save("output.xml")
```

Open the output file and notice that for every title for which Marjorie Green was an author, any co-authors have been removed and Stearns MacFeather has been added. The other title records were left untouched.

## Validation

Earlier, you learned how easily you can validate an XML document while reading it in by using the `ValidationType` property of the `XmlReaderSettings` class. Here, we'll look at how you can validate an XML document that is being created while it is still in memory. This is accomplished by using the `Validate` method exposed by the `XmlDocument` class.

### **XmlSchemaSet**

Before we can look at validating XML output, we must first look at another new class in the 2.0 Framework: the `XmlSchemaSet` class. In the 1.0 Framework, schemas were loaded into an `XmlSchemaCollection`, which was used for storing schemas used by the `XmlValidatingReader`. In the 2.0 Framework, both the `XmlSchemaCollection` and `XmlValidatingReader` have been retired. This accomplishes a number of things, including the following:

- ❑ **Retiring support for the Microsoft XDR format**—The `XmlSchemaSet` only supports W3C XML Schemas now. The `XmlSchemaCollection` also supported the proprietary XDR format.
- ❑ **Improving performance by reducing the number of compiles**—The `XmlSchemaCollection` would perform a compile after each schema was added. With the `XmlSchemaSet`, a single compile occurs by manually calling the `Compile` method after all of the schemas have been added.
- ❑ **Elimination of schema islands**—The `XmlSchemaCollection` improperly handled multiple schemas by treating them as separate “islands,” making all imports and includes only scoped to that particular schema. The `XmlSchemaSet` adds any imported schema to the schema set and treats the whole set as one logical schema.
- ❑ **Support of multiple schemas for a single namespace**—With the `XmlSchemaCollection`, each namespace could have only one schema. The `XmlSchemaSet` supports multiple schemas for the same namespace as long as there are not any type conflicts.

### **Validating Input and Output**

Let's start by loading the code from the previous example. In the declarations section, declare a new `XmlSchemaSet` and `ValidationEventHandler` object. We'll look at the `XmlSchemaSet` more closely in a minute. Add `pubs.xsd` to the schema set and call the `Compile` method. After loading the `XmlDocument`, add the schema set to the document. Then set the handler object to a new `ValidationEventHandler` instance pointing to a new method called `ValidationCallback` that you will create in a moment. This method will log the validation errors to a `TextBox` on the form. Finally call the `Validate` method of the `XmlDocument` object, passing in the handler to validate the input document and log the errors.

The code to update the XML document is the same as in the previous example. When the changes are complete, call the `Validate` method of the `XmlDocument` object again, passing in the handler. This will allow you to validate the output document before saving it to disk. Optionally you could create a second handler to use a different method to log the validation errors to the changed document rather than the original document. Finally you need to create the `ValidationCallback` method and add code to append the contents of the `Message` property of the `ValidationEventArgs` to the `TextBox`. Your code should look like the following:

## Chapter 5

```

Private Sub Button1_Click(ByVal sender As Object, ByVal e As EventArgs)
    Dim doc As System.Xml.XmlDocument
    Dim navigator, navigator2 As System.Xml.XPath.XPathNavigator
    Dim writer As System.Xml.XmlWriter
    Dim schemaSet As System.Xml.Schema.XmlSchemaSet
    Dim handler As System.Xml.Schema.ValidationEventHandler

    schemaSet = New System.Xml.Schema.XmlSchemaSet()
    schemaSet.Add(Nothing, "pubs.xsd")
    schemaSet.Compile()

    doc = New System.Xml.XmlDocument()
    doc.Load("pubs.xml")
    doc.Schemas = schemaSet
    handler = New System.Xml.Schema.ValidationEventHandler(AddressOf _
        ValidationCallback)

    TextBox1.Text += "Validating Input:" + System.Environment.NewLine
    doc.Validate(handler)

    For Each navigator In doc.CreateNavigator().Select( _
        "/pubs/titles[authors/@au_lname='Green']")

        navigator2 = navigator.SelectSingleNode("authors[@au_lname!='Green']")
        If Not IsNothing(navigator2) Then
            navigator2.DeleteSelf()
        End If

        writer = navigator.AppendChild()
        writer.WriteStartElement("authors")
        writer.WriteAttributeString("au_lname", "MacFeather")
        writer.WriteAttributeString("au_fname", "Stearns")
        writer.Close()
    Next

    TextBox1.Text += "Validating Output:" + System.Environment.NewLine
    doc.Validate(handler)
    doc.Save("output.xml")
End Sub

Public Sub ValidationCallback(ByVal sender As Object, ByVal e As _
    System.Xml.Schema.ValidationEventArgs)

    TextBox1.Text += e.Message + System.Environment.NewLine
End Sub

```

If you run this, you'll notice that you receive the exact same results as the previous example. This is because the document is valid, so the validation callback never gets called. To see the validation in action, simply change one of the line's output by the writer to output an invalid value, such as the author's last name:

```

writer.WriteAttributeString("au_lname2", "MacFeather")

```

## ADO.NET Integration with XML

Run the code again and you'll see two error messages stating the following: The 'au\_lname2' attribute is not declared. You can also change the original pubs.xml document to cause the input validation to fail, which will be caught by the first `Validate` method.

### Schema Inference

Validating the XML output of your code is very important in most situations. Unfortunately, you don't always have an XSD document to validate against. When this is the case, you can do the next best thing and allow the Framework to infer an XSD schema from an existing XML document. This is relatively simple to do now using the `XmlSchemaInference` class — as simple as replacing the following two lines of code:

```
schemaSet = New System.Xml.Schema.XmlSchemaSet()  
schemaSet.Add(Nothing, "pubs.xsd")
```

You need to first declare a new inference object and an `XmlReader`. At the location where you remove the two preceding lines, add a line to load the `XmlReader` from the same `pubs.xml` file the `XmlDocument` is reading. Set the inference object to a new instance of the `XmlSchemaInference` class and populate the schema set by calling the `InferSchema` method of the `XmlSchemaInference` object. Close your reader. You now have a schema you can use to validate your new XML against. You should receive the same results as the previous example after substituting the two lines above with the following lines:

```
Dim reader As System.Xml.XmlReader  
Dim inf As System.Xml.Schema.XmlSchemaInference  
reader = System.Xml.XmlReader.Create("pubs.xml")  
inf = New System.Xml.Schema.XmlSchemaInference()  
schemaSet = inf.InferSchema(reader)  
reader.Close()
```

### Change Notification

Whenever changes are made to the `XmlDocument`, events are raised to which you can add handlers to perform custom actions. Examples might be altering the values of an item being inserted, sending notifications when an item is deleted, and so on. Six events are provided for handling these scenarios. The first three are `ItemDeleting`, `ItemInserting`, and `ItemUpdating`. All three of these fire before the change has taken place. They are useful for actions such as performing validation of the data before allowing the change to take place. The other three are `ItemDeleted`, `ItemInserted`, and `ItemUpdated`. When these events fire, it is too late to modify the data, but they are very useful for actions such as logging. The following code builds upon the preceding example. You need to make the document declaration global and add two new methods: one for `ItemDeleted` and another for `ItemInserted`. In each of these, add a line to display the `OldValue` of each node that was deleted or the `NewValue` of the item that was inserted:

```
Dim WithEvents doc As System.Xml.XmlDocument  
  
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As _  
    System.EventArgs) Handles Button1.Click  
  
    Dim editor, editor2 As System.Xml.XPath.XPathEditableNavigator  
    Dim writer As System.Xml.XmlWriter
```

## Chapter 5

---

```
doc = New System.Xml.XmlDocument()
doc.Load("pubs.xml")

For Each navigator In doc.CreateNavigator().Select( _
    "/pubs/titles[authors/@au_lname='Green']")

    navigator2 = navigator.SelectSingleNode("authors[@au_lname!='Green']")
    If Not IsNothing(navigator2) Then
        navigator2.DeleteSelf()
    End If

    writer = navigator.AppendChild()
    writer.WriteStartElement("authors")
    writer.WriteAttributeString("au_lname", "MacFeather")
    writer.WriteAttributeString("au_fname", "Stearns")
    writer.Close()
Next

doc.Save("output.xml")
End Sub

Public Sub doc_ItemDeleted(ByVal sender As Object, ByVal e _
    As System.Xml.XmlNodeChangedEventArgs) Handles doc.NodeRemoved

    TextBox1.Text += "Deleted Item: " + e.OldValue + System.Environment.NewLine()
End Sub

Public Sub doc_ItemInserted(ByVal sender As Object, ByVal e _
    As System.Xml.XmlNodeChangedEventArgs) Handles doc.NodeInserted

    TextBox1.Text += "Inserted Item: " + e.NewValue + System.Environment.NewLine()
End Sub
```

## XSLT Improvements

As stated at the beginning of this chapter, one of the major design goals for this release of `System.Xml` is improved performance. The `XslCompiledTransform` class has been completely rewritten with this goal in mind. The 1.0 release of this class was written primarily based on MSXML 3.0. It performed pretty well, but there was still a lot of room for improvement. Since then, MSXML 4.0 has been released, which introduced several new optimizations to improve performance that the 2.0 framework benefits from.

The primary way it does this is by compiling the XSLT into IL code and using the Just In Time compiler to compile and run the IL against the XML document. Doing this means it takes a little bit longer to compile the XSLT stylesheet, but it runs much faster. In addition, by explicitly compiling the XSLT stylesheet and then running it against the source document, you can compile it a single time and run it repeatedly. This greatly improves performance when you need to transform numerous documents. The following code shows how to create an `XslCompiledTransform`, compile the stylesheet, and execute the conversion against an XML document:

```
Dim xslt As System.Xml.Query.XsltCommand

xslt = New System.Xml.Query.XsltCommand()
xslt.Compile("pubs.xslt")
xslt.Execute("pubs.xml", "output.html")
```

## Performance

All of the features we've looked at so far have made working with XML much easier, but probably few would disagree that the greatest improvement for XML in the 2.0 Framework is raw performance. We already described the efforts made to improve performance in the previous sections, such as minor tweaks made to optimize the most common code paths in all of the XML classes. You also saw how the XSLT processor has been completely rewritten to provide far better performance. You can see the result of all these efforts in Figure 5-3.

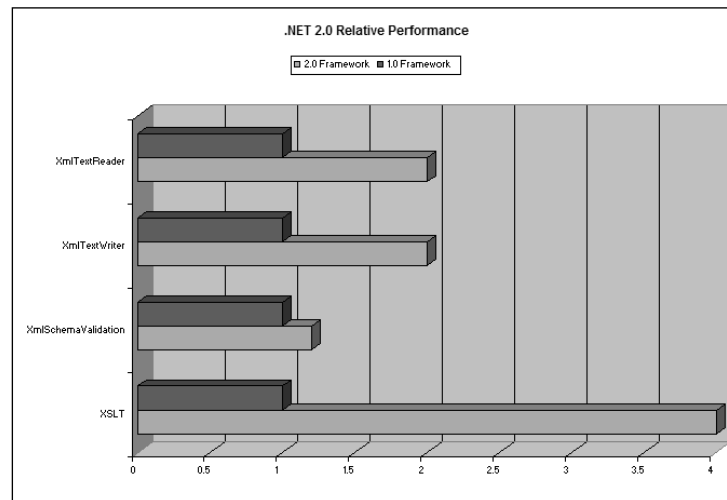


Figure 5-3

## Where XML Is Heading

It's hard to know for sure where XML or any rapidly advancing technology is going to be in as little as five years from now. The examples in this section describe some of the features and standards still in development and a sample of what we're likely to see in the next few years. These features are not included in the .NET 2.0 release but will likely be included in the next version.

## Chapter 5

---

### **XPath 2.0**

One of the standards that is right around the corner but wasn't finalized in time to be added to the .NET 2.0 Framework is XPath 2.0. For those who aren't already familiar with XPath, it is a query language for XML that is designed for selecting a subset of the XML document. It uses a syntax very similar to a file path, but with the capability to perform filters, calculate aggregates, and carry out other functions within it.

XPath 2.0 expands on these capabilities and introduces some new ones. The biggest change you'll notice is that the syntax has moved away from a simple path definition to what is known as a FLWOR (pronounced "flower") statement. FLWOR stands for For, Let, Where, Order, Return. It's very similar to the Select, Where, and Order By clauses of a SQL statement. By moving to this syntax, it is much easier to get the output you desire because it enables you to create and nest loops, store values in variables and manipulate them, sort the results, and define the exact output format.

### **XSLT 2.0 and XQuery**

XSLT 2.0 and XQuery are two new emerging formats that are supersets of XPath 2.0. They are somewhat competing technologies, and currently it is looking like XQuery is going to become the new standard for querying and manipulating XML data.

One advantage XSLT 2.0 has is that it is built upon the XSLT 1.0 standard that many developers are currently using and have grown attached to. XSLT 2.0 does not have a lot of radical changes planned other than support for XPath 2.0 statements. It is also XML-based, which can be a plus for dynamically generating queries.

XQuery, on the other hand, is not XML-based, although there is an XML form of it called XQueryX. The XQuery syntax is much easier for most people to read, and with the current definition, it can perform all of the same tasks as XSLT. XQuery offers one huge advantage, though: It can be used to query virtualized data sources (that is, data sources such as SQL Server or objects in memory that are not stored as XML by default but can be converted to XML on demand). Due to these advantages, Microsoft's plans at the time this book was written are to focus on providing XQuery support and to continue to maintain XSLT 1.0 support but not build XSLT 2.0 support in the .NET Framework.

### **XML Views**

One problem when working with XML is that, typically, the data source from which you retrieve the original data, or to which you want to save the data, is something other than XML, such as a SQL server. Currently, you have to write code to query the database, parse the results, and build the XML. If you plan to alter the XML document, you need to track the changes and then write code to perform the necessary update, inserts, and deletes to the original data store. If this is an operation that is going to be used frequently, you may choose to write this code as an XML Provider. Regardless of your approach, you still have to write the code.

Wouldn't it be nice if there were a generic component that could do this for you? That is exactly what XML Views are designed to do. They use declarative maps to transform the data instead of requiring custom code for each application. You create three maps: The first map is the XSD schema of the resulting XML you would like to work with. The second map defines your data source with information such

## ADO.NET Integration with XML

as the tables and fields you will use and their primary and foreign keys. The third map links the other two maps together and provides the field-to-field mapping between them.

Now that all of these maps are defined, the XML View has everything it needs to enable you to query the original data store using XQuery and return the results as XML. It will also track any changes made to the XML document and can persist them back to the data store—all of this without requiring you to write the custom conversion code. As an added bonus, because the maps are in a declarative format, they can be changed without recompiling the application.

### ObjectSpaces

ObjectSpaces is what every developer can't wait to get their hands on. It will allow you to declare your business objects in code and create maps very similar to XML Views that allow you to relate your business objects to database objects. You can probably guess what this means. No more writing data access code! Of course it won't be applicable in all situations, but it should significantly reduce the amount of code that is necessary to write for standard business applications. Unfortunately ObjectSpaces is still a ways away. It is currently planned for the next version of Visual Studio, currently codenamed Longhorn.

### Summary

It is hoped that you can see how the many new features for working with XML in .NET 2.0 can save you time and improve the performance, scalability, and security of your applications. XML is still rapidly developing and new uses are found for it every day. The expanding use of XML as a standard means of sharing data between application shows no sign of slowing down. XML Web Services are continuing to evolve. Emerging technologies described in the previous section are going to expand the implementation of XML within the .NET Framework, and the current enterprise architecture techniques continue a trend toward Service Oriented Architecture (SOA). All of these factors will help ensure the future expansion of XML—and along with it, the need to be able to efficiently work with XML. The new features in this release of the Framework and in future versions will continue to make it easier to perform everyday tasks related to XML.

### For More Information

- Microsoft XML Developer Center**—<http://msdn.microsoft.com/XML/default.aspx>
- MSDN .NET Framework Class Library (System.XML Namespace)**—<http://msdn.microsoft.com/library/en-us/cpref/html/frlrfssystemxml.asp>
- Arpan Desai's Weblog**—<http://blogs.msdn.com/arpande/>