1

What's New in Version 2.0 of the .NET Framework for XML

COPYRIGHTED MATERIAL

You are probably saying to yourself, "Whoa, wait a minute, I thought this book was about XML technology in SQL Server 2005." Yes, that is true. So why start the book off with a chapter about the XML technology found in version 2.0 of the .NET Framework?

Since the inception of the .NET Framework, Microsoft has taken a serious approach to supporting XML, a fact proven by looking at the amount of functionality provided in the System.Xml namespace, a group of classes specifically designed for the reading, writing, and updating of XML. Even in the first version of the .NET Framework, the support for XML was tremendous. The list of supported XML functionality included, but was not limited to, the following:

- Integration with ADO.NET
- Compliance with W3C standards
- Data source querying (XQuery)
- XML Schema support
- Ease of use

Microsoft set out to create a technology that dealt with data access using XML. Users of System.Xml in version 1.x of the .NET Framework agree that, on the whole, the technology contained a great number of useful classes that made dealing with XML and its related technologies a delight.

Even with all of the great advantages with version 1.1, it was not without its shortcomings. First and foremost, performance was an issue. Because of the way XML is processed, any obstacle or holdup in processing had a performance effect on the rest of the application. Security was another issue. For example, in the XML 1.0 specification, no precaution was taken to secure XML, which led to Denial of Service attacks via DTDs. Not good. The `XmlTextReader` had its own problems in that it could be subclassed and run in semitrusted code.

The inclusion of the CLR (Common Language Runtime) in SQL Server 2005 further strengthens the importance of understanding the XML technology from both sides, server and client. While the primary focus of this book is the support of XML in SQL Server 2005, a small handful of chapters focus on uncovering and understanding XML support in version 2.0 of the .NET Framework, and more important, how to utilize this technology in conjunction with SQL Server 2005 XML to get the most power and efficiency out of your application.

The entire goal of XML in version 2.0 of the .NET Framework boils down to a handful of priorities, with performance and W3C compliance at the top of the list. These are immediately followed by topics such as ease of use, or *pluggable*, meaning that the components are based on classes in the .NET Framework that can be easily substituted. Also included in the list is tighter integration with ADO.NET, which allows for datasets to read and write XML using the `XmlReader` and `XmlWriter` classes.

This chapter outlines some of the major feature enhancements made to the System.xml namespace in version 2.0 of the .NET Framework. If you look at all the changes made to the System.xml namespace, that list could possibly take up a very large portion of a book. The goal of this chapter, however, is to highlight the handful of significant changes that you will most likely use on a day-to-day basis to help improve your XML experience.

System.xml Version 2.0 Enhancements and New Features

The following list contains the System.xml enhancements that are covered in this chapter:

- Performance
- Type support
- XPathDocument
- XPathEditableNavigator
- XML query architecture
- `XmlReader`, `XmlWriter`, and `XmlReaderSettings`

Ideally, this list would include XQuery support. Unfortunately, in a January 2005 MSDN article, Microsoft announced that it would be pulling client-side XQuery support in version 2.0 of the .NET Framework. While the pains of realization set in, their reasons are justifiable. The main reason for pulling XQuery support was for the simple reason of timing. XQuery has yet to become a W3C recommendation and since it has not yet, this opens XQuery up for some changes. This put Microsoft in the peculiar situation of trying to meet the requests of its customers while trying to keep with future compatibility. Microsoft did not want to support a technology that could possibly change. That is not to say, however, that you won't ever see support for client-side XQuery. Microsoft's goal is to add it back in once XQuery has reached recommendation — which I hope will happen quickly.

Time to dig right in. The following section deals with arguably the most important enhancement to version 2.0 of the .NET Framework: performance.

Performance

You have to admit that developers like it when things go fast, and the faster the better. Developers absolutely hate waiting. XML performance is no different. This section, then, discusses the places where Microsoft focused the majority of the performance improvements. There isn't any code in this section to try out, but feel free to run some performance tests using some of the concepts discussed in this section.

XMLTextWriter and XMLTextReader

To begin with, the `XMLTextWriter` and `XMLTextReader` have been significantly re-written to cut these two call times nearly in half. Both of these classes have been completely rewritten to use a common code path.

XMLReader and XMLWriter

The `XmlReader` and `XMLWriter` classes can now be created via the `Create` method. In fact, they outperform the `XmlTextReader` and `XmlTextWriter` and as is discussed a little bit later, the `Create` method is now the preferred method of reading and writing XML documents.

XSLT Processing

XSLT processing performance has dramatically increased in version 2.0 of the .NET Framework. To understand why, you need to understand the `XslTransform` class. The `XslTransform` class, found in the System.Xml.Xsl namespace, is the brains behind XSLT. Its job is to transform the contents of one XML document into another XML document that is different in structure. The `XslTransform` class *is* the XSLT processor.

In version 1.1 of the .NET Framework, the `XslTransform` class was based on version 3.0 of the MSXML XSLT processor. Since then, version 4.0 of the MSXML XSLT processor came out and included enhancements that vastly improved the performance of the XSLT processor. So what's up with version 2.0 of the .NET Framework?

The idea with version 2.0 of the .NET Framework was to improve better yet the XSLT processing beyond that of the MSXML 4.0 XSLT processor. In order to do this, Microsoft completely rebuilt the XSLT processor from the ground up. The new processor is now called the `XslCompileTransform` class and lives in the System.Xml.Xsl namespace.

This new class has the same query runtime architecture as does the CLR, which means that it is compiled down to intermediate format at compile time. There is an upside and downside to this. The downside is that it will take longer to compile your XSLT style sheet. The upside is that the runtime execution is much faster.

Because there is no XQuery support at this time, performance improvements in the `XslCompileTransform` class are critical since XML filter and transformation still need to use XSLT and XPath. To help with this, Microsoft added XSLT debugger support in Visual Studio 2005 to debug style sheets. This comes in handy.

XML Schema Validation

There is one major reason why XML Schema validation performance has improved, and that is type support. Type support will be defined in more detail in the next section; however, for XML Schema validation, type support comes into play in a huge way when you try to load or transform an XML document.

When an XML document is loaded into a reader and a schema applied to it, CLR types are used to store the XML. This is useful because xs:long is now stored as a CLR long. First, the XML stores better this way. Second, there's no more of this useless untyped string stuff.

Type support also applies when creating an `XPathDocument` by applying XSLT to an original `XPathDocument`. In this scenario, the types are passed from one document to another without having to copy to an untyped string and then reparse them back the original type. This in itself is a tremendous performance boost, especially when linking multiple XML components.

Conversion between schema types and CLR types was possible in version 1.1 using the `XmlConverter` helper class, but conversion support is now extended to any `XmlReader`, `XmlWrite`, and `XPathNavigator` class, discussed in the next section.

Type Support

While XQuery support has been removed from version 2.0 of the .NET Framework, type support for many of the XML classes now offers type conversions. Classes such as the `XmlReader`, `XmlWrite`, and `XPathNavigator` are all now type-aware, and support conversion between CLR types and XML schema types.

In version 1.0 of the .NET Framework, type conversion was done by using the `xmlConvert` method, which enabled the conversion of a schema data type to a CLR (or .NET Framework) data type.

For example, the following code demonstrates how to convert an `xml` string value to a CLR Double data type using the `XmlConvert` in version 1.0 of the .NET Framework:

```
Imports System.Xml

 'declare local variables
Dim xtr As XmlTextReader = New XmlTextReader("c:\testxml.xml")
Dim SupplierID As Integer

'loop through the xml file
Do While xtr.Read()
    If xtr.NodeType = XmlNodeType.Element Then
        Select Case xtr.Name
            Case "SupplierID"
                SupplierID = XmlConvert.ToInt32(xtr.ReadInnerXml())
        End Select
    End If
Loop
```

While converting an untyped value of an XML node to a .NET Framework data type is still supported in version 2.0 of the .NET Framework, you can accomplish this same thing via a single method call new to version 2.0 of the .NET Framework. Using the `ReadValueAs` method call provides improved performance (because of the single method call) and is easier to use.

For example, you could rewrite the previous code as follows:

```
Imports System.Xml

  'declare local variables
Dim xtr As XmlTextReader = New XmlTextReader("c:\testxml.xml")

Dim SupplierID As Integer

 'loop through the file
 Do While xtr.Read()
     If xtr.NodeType = XmlNodeType.Element Then
         Select Case xtr.Name
             Case "SupplierID"
                 SupplierID = xtr.ReadElementContentAsInt()
         End Select
     End If
 Loop
```

The same principle can be applied to attributes and collections as well. For example, element values (as long as they are separated by spaces) can be read into an array of values such as the following:

```
Dim I as integer
Dim elementvalues() as integer = xtr.ReadValueAs(TypeOf(elementvalues())
For each I in elementvalues()
 Console.WriteLine(i)
Next I
```

So far the discussion has revolved around untyped values, meaning that all the values have been read from the XML document and stored as a Unicode string value that are then converted into a .NET Framework data type.

An XML document associated with an XML schema through a namespace is said to be *typed*. Type conversion applies to typed XML as well because the types can be stored in the native .NET Framework data type. For example, xs:double types are stored as .NET Double types. No conversion is necessary; again, improving performance.

All the examples thus far have used the `XmlReader`, and as much fairness should be given to the `XmlWriter` for Type conversion, which it has. The new `WriteValue` method on the `XmlWriter` class accomplishes the same as the `ReadValueAs` does for the `XmlReader` class.

In the following example, the `WriteValue` method is used to write CLR values to an XML document:

```
Imports System.Xml

Dim BikeSize As Integer = 250
Dim Manufacturer As String = "Yamaha"
```

```
Dim xws As XmlWriterSettings = New XmlWriterSettings
xws.Indent = True
Dim xw As XmlWriter = XmlWriter.Create("c:\motocross.xml", xws)
xw.WriteStartDocument()
xw.WriteStartElement("Motocross")
xw.WriteStartElement("Team")
xw.WriteStartAttribute("Manufacturer")
xw.WriteValue(Manufacturer)
xw.WriteEndAttribute()
xw.WriteStartElement("Rider")
xw.WriteStartAttribute("Size")
xw.WriteValue(BikeSize)
xw.WriteEndAttribute()
xw.WriteElementString("RiderName", "Tim Ferry")
xw.WriteEndElement()
xw.WriteEndElement()
xw.WriteEndDocument()
xw.Close()
```

Running this code produces the following results in the `c:\testmotocross.xml` file:

```
<?xml version"1.0" encoding="utf-8" ?>
<Motocross>
 <Team Manufacturer="Yamaha">
         <Rider Size="250">
                <RiderName>Tim Ferry</RiderName>
         <Rider>
 </Team>
</Motocross>
```

Now that a lot of the XML classes are type-aware, they are able to raise the schema types with additional conversion support between the schema types and their CLR type counterparts.

XPathDocument

The `XPathDocument` was included in version 1 of the .Net Framework as an alternative to the DOM for XML Document storage. Built on the XPath data model, the primary goal of `XPathDocument` was to provide efficient XSLT queries.

If the purpose of the `XPathDocument` is for XML Document storage, then what happened to the DOM? The DOM is still around and probably won't be going away any time soon. However, there are reasons why an alternative was necessary. First, the acceptance of XML is moving at an extremely fast rate, much faster than the W3C can keep up with the DOM recommendations. Second, the DOM was never really intended for use with XML as a data storage facility, specifically when trying to query the data. The DOM was created at the time when XML was just being adopted and obtaining a foothold in the development communities. Since then, XML acceptance has accelerated greatly and the DOM has not made the adjustments necessary to keep up in improvements. For example, XML documents are reaching high levels of capacity and the DOM API is having a hard time adapting to these types of enterprise applications.

Basically, the DOM has three shortcomings. First, the DOM API is losing its hold on the XML neighborhood with the introduction of `XmlReader` and `XmlWriter` as ways to read and write XML documents. Most developers are ready to admit that the DOM is not the friendliest technology to grasp. The `System.Xml` class provided an easy way to read and write XML documents. Second, the DOM data model is based on XML syntax and query language syntax is not. This makes for inefficient XML document querying. Lastly, application modifications are a must when trying to find better ways to store XML in the application. This is primarily due to the fact that there is no way to store XML documents. Version 2.0 of the .NET Framework has greatly improved the `XPathDocument` by building on better query support and `XPathNavigator` API found in version 1.

The goal of the `XPathDocument` in version 2.0 was to build a much better XML store. To do that, a number of improvements were made, including the following:

- `XmlWriter` to write XML content
- Capability to load and save XML documents
- Capability to accept or reject XML document changes
- XML store type support

What you will find is that the `XPathDocument` has all of the capabilities of the `XmlDocument` class with the added features of great querying functionality. On top of that, you can work in a disconnected state and track the changes made to the XML document.

The next section includes a number of examples to demonstrate loading, editing, and saving XML documents.

XPathNavigator

The `XPathNavigator` class provides a mechanism for the navigation and editing of XML content and providing methods for the editing of nodes in the XML tree.

> *In version 1.1 of the .NET Framework, the* `XPathNavigator` *class was based purely on version 1.0 of the XPath data model. In version 2.0 of the .NET Framework, the* `XPathNavigator` *class is based on the XQuery 1.0 and XPath 2.0 data models.*

As part of the `System.Xml.XPath` namespace, the XPathNavigator class allows for very easy XML document navigation and editing. Using the XML document example created previously, the following code loads that XML document and appends a new `Rider` element using the `XmlWriter` and `XPathNavigator` classes:

```
Dim xpd as XPathDocument = New XPathDocument("c:\motocross.xml")
Dim xpn as XPathDocument = xpd.CreateNavigator

Xpen.MoveToFirstChild()
Xpen.MoveToNext()

Using xw As XmlWriter = xpn.AppendChild
 xw.WriteStartElement("Bike")
```

```
    xw.WriteAttributeString("Size", "250")
    xw.WriteElementString("RiderName", "Chad Reed")
    xw.WriteEndElement()
    xpd.Save("c:\motocross.xml")
   End Using
```

The move from version 1.0 of XPath to version 2.0 is important for several reasons. First, there are better querying capabilities. For example, version 1.0 of XPath supported only four types, whereas version 2.0 supports 19 types. The second reason is better performance. XQuery 1.0 and XPath 2.0 nearly share the same foundation; XPath 2.0 is a very explicit subset of the XQuery 1.0 language. Because of this close relationship between the two, once you have learned one, you nearly understand the other.

XML Query Architecture

The XML query architecture provides the capability to query XML documents using different methods such as XPath and XSLT (with XQuery to be provided later). The classes that provide this functionality can be found in the `System.Xml.Xsl` namespace. Part of this functionality is the capability to transform XML data using an XSLT style sheet.

In version 2.0 of the .NET Framework, transforming XML data is accomplished by calling the `XslCompileTransform` class, which is the new XSLT processor. The `XslCompileTransform` class was mentioned previously during the discussion of performance. That section covered the topic of how the `XslCompileTransform` was created to improve XSLT performance. In this section, however, the focus of discussion will be on using the new XSLT processor and its associated methods.

The `XslCompileTransform` class replaces the `XslTransform` class in version 1.0 of the .NET Framework. Therefore, it is needless to say that the `Load` and `Transform` methods of the `XslTransform` class are also obsolete. What replaces them? The `XslCompileTransform` is very similar in architecture to the `XslTransform` class in that it also has two methods: the `Compile` method and the `Execute` method.

The `Transform` method of the `XslCompileTransform` class does exactly what the `Compile` method of the `XsltCommand` class did: it compiles the XSLT style sheet specified by the overload parameter. For example, the following code compiles the style sheet specified by the `XmlReader`:

```
Dim ss as String = "c:\motocross.xsl")
Dim xr as XmlReader = XmlReader.Create(ss)
Xr.ReadToDescendant("xsl:stylesheet")
Dim xct as XslCompiledTransform = new  XslCompiledTransform
 xct.Transform(xw)
```

In this example, you create the `XmlReader`, and then use its `ReadToDescendant` property to advance the `XmlReader` to the next descendant element using the qualified name. The `XslCompileTransform` is then created and the `Transform` method is called with the Reader.

The next step is to call the `Execute` method to execute the transform using the compiled style sheet. Using the previous example, add the following code:

```
Dim ss as String = "c:\motocross.xsl")
Dim xr as XmlReader = XmlReader.Create(ss)
Xr.ReadToDescendant("xsl:stylesheet")
Dim xct as XslCompileTransform = new  XslCompileTransform
 xct.Transform(xw)
Dim xpd as XPathDocument = New XPathDocument("c:\motocross2.xml")
Dim xw as XmlWriter = XmlWriter.Create(Console.Out)
Xs.Execute(New XPathDocument("c:\motocross2.xml"), xw)
Xw.close
```

The `Execute` method takes two input types for the source document: the `IXPathNavigatable` interface or a string URI.

The `IXPathNavigatable` interface is implemented in the `XmlNode` or `XPathDocument` classes and represents an in-memory cache of the XML data. Both classes provide editing capabilities.

The other option is to use the source document URI as the XSLT input. If this is the case, you will need to use an `XmlResolver` to resolve the URI (which is also passed to the `Execute` method).

Transformations can be applied to an entire document or a node fragment. However you're transforming a node fragment, you need to create an object containing the node fragment and pass that object to the `Execute` *method.*

XmlReader, XmlReaderSettings, XmlWriter, and XmlWriterSettings

Throughout this chapter you have seen a number of examples of how to use the `XmlReader` and `XmlWriter` classes. This section highlights a number of new methods that complement the existing methods of both of these classes.

The static `Create` method on both the `XmlReader` and `XmlWriter` classes is now the recommended way to create `XmlReader` and `XmlWriter` objects. The `Create` method provides a mechanism in which features can be specified that you want both of these classes to support.

As seen previously, when combined with the `XmlReaderSettings` class, you can enable and disable features by using the properties of the `XmlReaderSettings`, which are then passed to the `XmlReader` and `XmlWriter` classes.

By using the `Create` method together with the `XmlReaderSettings` class, you get the following benefits:

- You can specify the features you want the `XmlReader` and `XmlWriter` objects to support.
- You can add features to existing `XmlReader` and `XmlWriter` objects. For example, you can use the `Create` method to accept another `XmlReader` or `XmlWriter` object and you don't have to create the original object via the `Create` method.
- You can create multiple `XmlReaders` and `XmlWriters` using the same settings with the same functionality. The reverse of that is also true. You can also modify the `XmlReaderSettings` and create new `XmlReader` and `XmlWriter` objects with completely different feature sets.

- You can take advantage of certain features only available on `XmlReader` and `XmlWriter` objects when created by the `Create` method, such as better XML 1.0 recommendation compliance.
- The `ConformanceLevel` property of the `XmlWriterSettings` class configures the `XmlWriter` to check and guarantee that the XML document being written complies with XML rules. Certain rules can be set so that, depending on the level set, you can check the XML document to make sure it is a well-formed XML document. There are three levels:
  - **Auto:** This level should be used only when you are absolutely sure that the data you are processing will always be well-formed.
  - **Document:** This level ensures that the data stream being read or written meets XML 1.0 recommendation and can be consumed by any XML processor; otherwise an exception will be thrown.
  - **Fragment:** This level ensures that the XML data meets the rules for a well-formed XML fragment (basically, a well-formed XML document that does not have a root element). It also ensures that the XML document can be consumed by any XML processor.

Reading this list, you would think that it couldn't get any better. To tell you the truth, there are additional benefits with some of the items. For example, in some cases when you use the `ConformanceLevel` property, it automatically tries to fix an error instead of throwing an exception. If it finds a mismatched open tag, it will close the tag.

It is time to finish this chapter off with an example that utilizes a lot of what you learned:

```
Dim BikeSize As Integer = 250
Dim Manufacturer As String = "Yamaha"
Dim xws As XmlWriterSettings = New XmlWriterSettings
xws.Indent = True
xws.ConformanceLevel = ConformanceLevel.Document
Dim xw As XmlWriter = XmlWriter.Create("c:\motocross.xml", xws)
xw.WriteStartDocument()
xw.WriteStartElement("Motocross")
xw.WriteStartElement("Team")
xw.WriteStartAttribute("Manufacturer")
xw.WriteValue(Manufacturer)
xw.WriteEndAttribute()
'First Rider
xw.WriteStartElement("Rider")
xw.WriteStartAttribute("Size")
xw.WriteValue(BikeSize)
xw.WriteEndAttribute()
xw.WriteElementString("RiderName", "Tim Ferry")
xw.WriteEndElement()
'Second Rider
xw.WriteStartElement("Rider")
xw.WriteStartAttribute("Size")
xw.WriteValue(BikeSize)
xw.WriteEndAttribute()
xw.WriteElementString("RiderName", "Chad Reed")
xw.WriteEndElement()
xw.WriteEndDocument()
xw.Close()
```

The preceding example creates an XML document and writes it to a file. That file is then reloaded, and using the `XPathEditableNavigator` and `XPathNavigator`, a new node is placed in the XML document and resaved.

Summary

Now that you have an idea of the new XML features that appear in version 2.0 of the .NET Framework, you should also understand why this chapter was included in the book. Microsoft is taking a serious stance on XML technology and it is really starting to show with a lot of the features covered in this chapter.

Performance in XML is imperative to overall application performance, so this was a great place to start. As discussed, many improvements were made in this area so that XML performance was not the bottleneck in application performance. You also spent a little bit of time looking at where those performance improvements were made, such as modifications to certain classes sharing the same code path and complete class re-writes.

You read about the new type support added to the `XmlReader`, `XmlWriter`, and `XmlNavigator` classes, which contributes to the overall performance of XML, but more important, makes it much easier to read and write XML without the headaches of data type conversions.

You will probably agree that the `XPathDocument` and `XPathEditableNavigation` were fun to read and put to test. This is some absolutely cool technology that will make working with XML much easier and a lot more fun than in the past as compared to the DOM. The DOM isn't going away, but these technologies are far better suited for XML storage.

The enhancements to the `XmlWriter`, `XmlReader`, `XmlReaderSettings`, and `XmlWriterSettings` are a welcomed improvement, as you learned how easy it is to read, write, and modify XML documents.

Last, the topic of XML query architecture was discussed, along with the new `XslCompiledTransform` class, which replaces the `XslTransform` class, as well as how to use the new methods on that class.

In the next chapter you discover what's new in SQL Server 2005 XML (which is why you bought the book, right?) and all the new XML support it provides.