Programmer to Programmer ™

ASP.NET Distributed Data Applications

Written and tested for final release of .NET v1.0

Alex Homer and Dave Sussman

Wrox technical support at: support@wrox.com
Updates and source code at: www.wrox.com
Peer discussion at: p2p.wrox.com

What You Need to Use This Book

The following list is the recommended system requirements for running the code in this book:

- Windows 2000 Professional or higher with IIS installed
- Windows XP Professional with IIS installed
- ASP.NET Version 1.0
- SQL Server 2000 or MSDE
- Visual Studio .NET Professional or higher (optional)

In addition, this book assumes the following knowledge:

- A good understanding of the .NET Framework and ASP.NET
- An understanding of the VB.NET language and JavaScript

Summary of Contents

Accessing XML Documents

In the previous chapter we looked at how we can expose data via components in our application's data tier in a range of formats. Depending on the features of our application's middle tier, or the type of client we are serving to, we may have to tailor the data format to suit their requirements. As you saw, this is relatively easy with the .NET data access classes included in the Framework.

However, all the examples in the previous chapter accessed a data source that is a relational data store – in our case a SQL Server database. What if the data source is not "relational", but (as is increasingly becoming the case) an XML document? It could be a file stored on disk, or a stream of XML delivered by another component, a Web Service, or another service or application such as BizTalk Server. Or it might simply be delivered over HTTP as a POST from a Web page.

In this chapter, we'll continue examining the examples we started looking at in the previous chapter. As well as a series of pages that use data extracted from a relational database, there is an equivalent set of pages that use data stored as an XML document in a disk file. So, the plan for this chapter is to show:

- How we can build components that access XML documents
- How we can convert and return the XML as various different object types
- How we can display the XML easily in our ASP.NET pages

We start with a look at the data access component that we are using throughout this chapter.

XML Data Access Component Examples

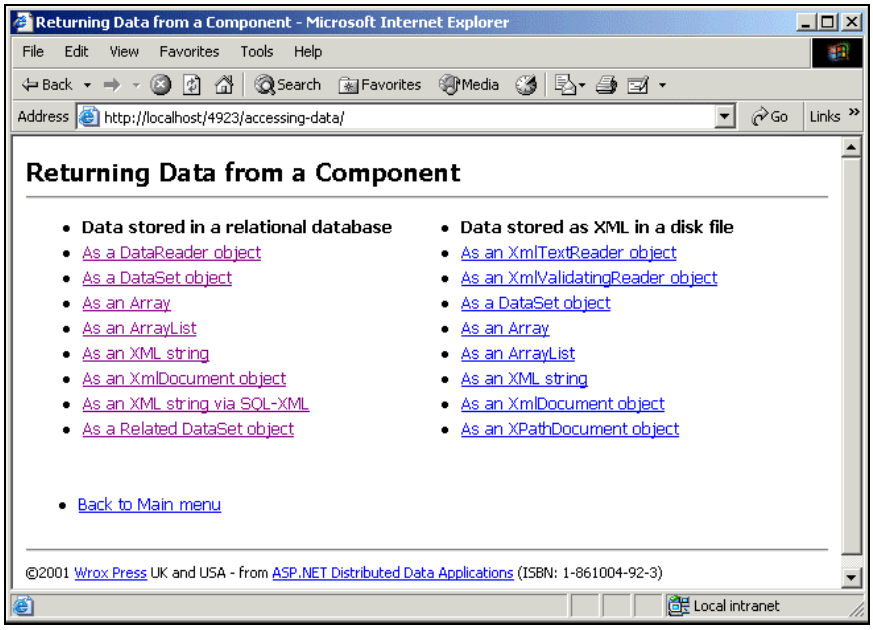
All the examples we looked at in the previous chapter accessed data stored in a relational database, predominantly using ADO.NET objects (such as the `DataReader` and `DataSet`) to extract the data, and returning it in various other formats as well – such as an `Array` and an XML document. However, there are an increasing number of occasions where the data is supplied in XML format. Obvious cases are when it is stored as a disk file, streamed from another component, or delivered to our application via a Web Service (using SOAP).

The client may require the data in a format suitable for data binding, such as a rowset of some kind. Or, as we've previously discussed, there are useful possibilities for using data as XML where the client is able to process this XML directly, for example using an XML parser located on the client machine. We demonstrate this in a range of ways in later chapters. Of course, if the .NET Framework is installed on the client, we can use the .NET `System.XML` classes there instead. However, in the remainder of this chapter we'll consider how we can take XML data and expose it to the client or the middle tier in a range of different ways.

> *You can download the samples shown here to run on your own systems. See the section "Setting Up the Examples" in the previous chapter for details of how to install and configure them on your machine.*

Accessing and Returning Relational Data

The example files we provide include a set of components whose source code is provided in the `vb-components` folder, and a series of pages stored in the `accessing-data` folder that use these components. This folder also contains the `default.htm` menu page (originally seen in the previous chapter) for the examples you'll see here:



The SupplierListXml Data Access Component

As we did in the previous chapter, we've provided one data access component for all the examples in this chapter. The component, `SupplierListXml.dll`, implements several methods that we use to expose the XML data in different ways. The source code of the component can be examined in the `vb-components/SupplierListXML` folder, or by following the links at the top of each example page (as demonstrated in the relational data access examples in the previous chapter).

The component imports the .NET classes it requires, in this case `System` and the three `System.Xml` namespaces. It defines a namespace for our component class, `Wrox4923Xml`, and several `Private` member variables that we want to be able to access from more than one function within the component:

```
Imports System
Imports System.Xml
Imports System.Xml.XPath
Imports System.Xml.Schema

'the namespace for the component
Namespace Wrox4923Xml

  'the main class definition
  Public Class SupplierListXml

    'private internal variables
    Private m_XmlFilePath As String       'path and name of XML file
    Private m_SchemaPath As String        'path and name of Schema file
    Private m_Validate As Boolean = False 'if validation to be performed

    'variable to hold instance of XmlTextReader object
    'allows it to be accessed by separate functions
    Private m_XTReader As XmlTextReader
```

The constructor function for the component comes next. It accepts two parameters – the first is mandatory and specifies the full path to the XML source file on disk that we're using as our data source. The second optional parameter is the full path to an XML Schema (XSD) file that we want the component to use to validate the XML document. If this parameter is omitted, no validation is performed.

```
    Public Sub New(ByVal XmlFilePath As String, _
                   Optional ByVal SchemaPath As String = "")
      MyBase.New() 'call constructor for base class
      m_XmlFilePath = XmlFilePath
      m_SchemaPath = SchemaPath
      If m_SchemaPath.Length > 0 Then m_Validate = True
    End Sub
```

However, note that three of the functions in the component **require** a schema to be provided, and these methods will raise an error if they are called when the optional second parameter to the constructor was not set when the component instance was created. The three methods are `GetSuppliersValidatingReader`, `GetSuppliersDataSet` and `GetSuppliersXmlString`.

Returning an XmlTextReader Reference

We start with a simple example, returning a reference to an open `XmlTextReader` object. Members of the `XmlReader` family of objects (`XmlTextReader` and `XmlNodeReader`) are the XML equivalent of the `DataReader` object we used in the previous chapter. They act like a "pipe" between our applications and an XML document (stream or disk file).

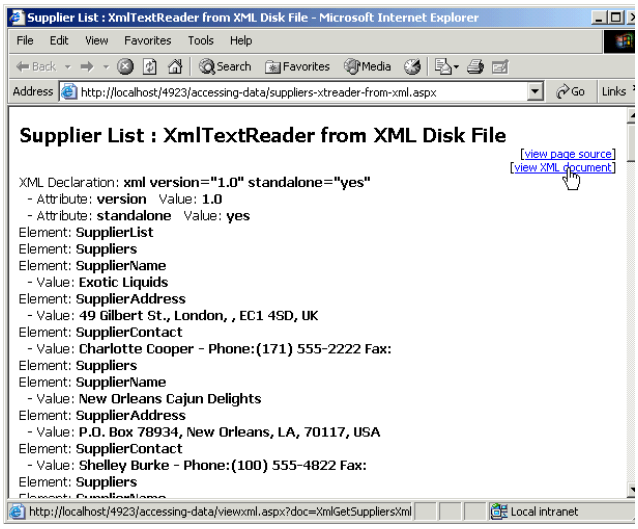
However, unlike the `DataReader`, they return XML elements rather than data rows and columns. We used an `XmlTextReader` in an example in the previous chapter to return the XML elements that the SQLXML technology built into SQL Server 2000 returns.

All we have to do to implement our `GetSuppliersXmlTextReader` method is to create and initialize an `XmlTextReader` object from a disk file. We specify the full physical path to the file in the `XmlTextReader` object's constructor, catch any error (for example, file not found) and raise this error to the calling routine. Otherwise we just pass back a reference to the `XmlTextReader`:

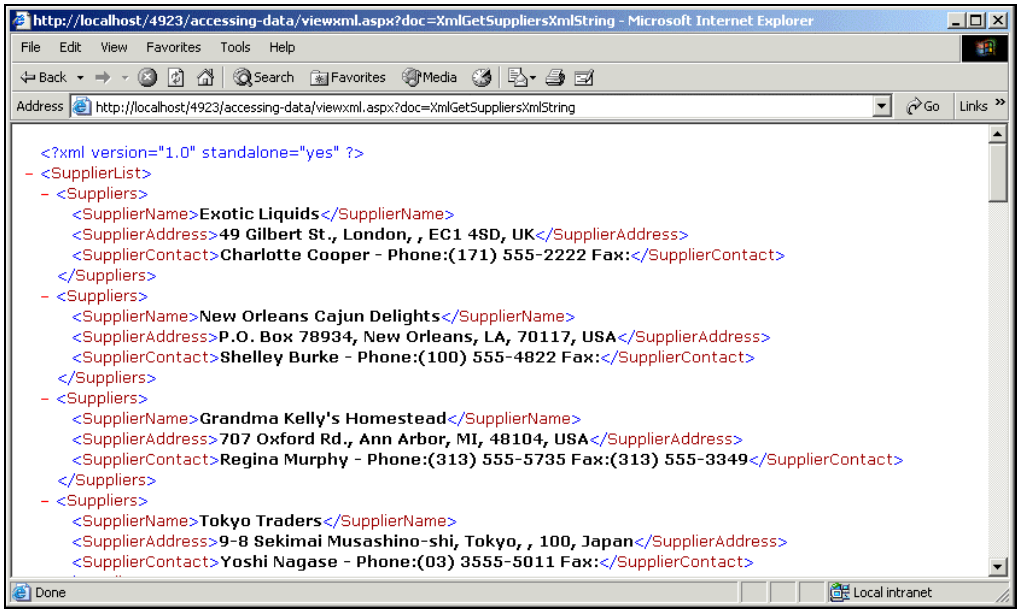
```
Public Function GetSuppliersXmlTextReader() As XmlTextReader

  Try

    'create new XmlTextReader object and load XML document
    Return New XmlTextReader(m_XmlFilePath)

  Catch objErr As Exception

    Throw objErr    'throw exception to calling routine

  End Try

End Function
```

Displaying the Results

The page `suppliers-xtreader-from-xml.aspx` uses this function to read the XML source document we're using in these examples. When you open it, the page displays a list of elements from the XML document:



There are links at the top of the page to view the source of the `.aspx` file (and from that screen, the component source code), and to display the XML document itself. The next screenshot shows a section of this XML document – you can see that it's the same as the XML that we returned from our relational database in some of the examples in the previous chapter:



The example page contains an `Import` directive for our component, as well as one for the `System.Xml` classes (we'll be using some of the types defined there in the code in our page):

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

It also contains an `asp:Label` control in which we display the results:

```
<asp:Label id="lblMessage" runat="server" />
```

The Page_Load Event Handler

The `Page_Load` event code is responsible for instantiating our component and calling the method that returns an `XmlTextReader`. The first step is to create the physical path to the XML file we're using – this section of code is the same in most of the pages you'll see in this chapter:

```
Sub Page_Load()

  'create physical path to XML file (in same folder as ASPX page)
  Dim strCurrentPath As String = Request.PhysicalPath
  Dim strPathOnly As String = Left(strCurrentPath, _
                              InStrRev(strCurrentPath, "\"))
  Dim strXMLPath As String = strPathOnly & "supplier-list.xml"
```

Now we declare a variable to hold the `XmlTextReader` returned by our method, create the component instance and call the `GetSuppliersXmlTextReader` method:

```
    'declare variable to hold XmlTextReader object
    Dim objReader As XmlTextReader

    Try

      'create an instance of the data access component
      Dim objSupplierList As New Wrox4923Xml.SupplierListXml(strXMLPath)

      'call the method to return the data as an XmlTextReader
      objReader = objSupplierList.GetSuppliersXmlTextReader()
```

Now we can use the `XmlTextReader` in our page. We want to display the XML it references, so we created a separate function within this page that reads and parses the XML and returns it as a `String`. The function `ParseXmlContent` takes the `XmlTextReader` and returns the content ready for display, so we can just assign it to our `asp:Label` control:

```
      'if we got a result, display the parsed content
      If Not objReader Is Nothing Then
        lblMessage.Text = ParseXmlContent(objReader)
      End If
```

We now have two more tasks to take care of. If there was an error using the component, we want to catch this and display it in the page, and finally, we must remember to close the `XmlTextReader` we used:

```
    Catch objErr As Exception

      'there was an error and no data will be returned
      lblMessage.Text = "ERROR: No data returned. " & objErr.Message

    Finally

      objReader.Close()   'close the Reader to release file

    End Try

  End Sub
```

This example page uses a custom function named `ParseXmlContent` to read the XML content direct from the `XmlTextReader` and return it as a string ready for display. We'll describe that function next – it is used in the next example in this chapter as well.

Parsing an XML Document

The `XmlReader` objects provided in the `System.Xml` namespace of the Framework provide a "pull" model for accessing XML documents in much the same way as the `DataReader` does for relational data (and unlike a SAX XML parser, which uses the "push" model and requires event handlers to be created to handle events as the document is read).

In many ways, the .NET "pull" model of accessing data is more intuitive and easier to use. The `XmlReader` objects have methods that allow us to read nodes from a document in a specific format, or retrieve them as node objects. As each node is read we can get information about the node, access the attributes of that node, or skip nodes.

The following listing of our `ParseXmlContent` function demonstrates many of these features. It reads nodes in turn from the document and checks to see what type they are – outputting appropriate information for the three common types we know exist in our document. Then it looks for any attributes and outputs information about these, before moving on to the next node:

```
Function ParseXmlContent(objXMLReader As XmlTextReader) As String

  'read or "pull" the nodes of the XML document
  Dim strNodeResult As String = ""
  Dim objNodeType As XmlNodeType

  'read each node in turn - returns False if no more nodes to read
  Do While objXMLReader.Read()

    'select on the type of the node (these are only some of the types)
    objNodeType = objXMLReader.NodeType

    Select Case objNodeType

      Case XmlNodeType.XmlDeclaration:
        'get the name and value
        strNodeResult += "XML Declaration: <b>" & objXMLReader.Name _
                       & " " & objXMLReader.Value & "</b><br />"

      Case XmlNodeType.Element:
        'just get the name, any value will be in next (#text) node
        strNodeResult += "Element: <b>" & objXMLReader.Name & "</b><br />"

      Case XmlNodeType.Text:
        'just display the value, node name is "#text" in this case
        strNodeResult += "  - Value: <b>" & objXMLReader.Value _
                       & "</b><br />"

    End Select

    'see if this node has any attributes
    If objXMLReader.AttributeCount > 0 Then

      'iterate through the attributes by moving to the next one
      'could use MoveToFirstAttribute but MoveToNextAttribute does
      'the same when the current node is an element-type node
      Do While objXMLReader.MoveToNextAttribute()

        'get the attribute name and value
        strNodeResult += "  - Attribute: <b>" & objXMLReader.Name _
                       & "</b>   Value: <b>" & objXMLReader.Value _
                       & "</b><br />"
      Loop

    End If

  Loop   'and read the next node

  'and return the resulting string
  Return strNodeResult

End Function
```

Returning an `XmlTextReader` object is therefore a useful way to provide our application with access to an XML document that arrives as a stream or is stored as a disk file. However, bear in mind that it exhibits the same pros and cons as the `DataReader` we used in the previous chapter to access relational data.

The `XmlReader` objects are **connected** data sources, and so require their "connection" (or reference) to the source data to be maintained all the time that the XML data is being read. However, they are lightweight, fast and efficient for situations where the "connected" limitation is acceptable.

On the other hand, when we want to remote data and work with it in a disconnected environment, we need to look for a different object to use. Later in this chapter we'll see some options – including using a `DataSet` and one of the `XmlDocument` family of objects.

Returning an XmlValidatingReader Reference

One regular requirement when reading XML is to validate it against a specific schema to ensure that the format is suitable for the application that is using it. It makes sense to do this within our data access layer components as we actually read the document from a stream or disk file.

Within the .NET `System.Xml` namespace classes, validation is usually performed through an `XmlValidatingReader`. We "attach" this to an `XmlReader` object in order to perform the validation as the document is read. Our example data access component contains a method named `GetSuppliersValidatingReader` that instantiates and returns an `XmlValidatingReader`. The following listing shows this method. You can see that it uses a separate function named `GetValidatingReader` within our data access component to actually get the `XmlValidatingReader` we return to the calling routine:

```
Public Function GetSuppliersValidatingReader() As XmlValidatingReader

  If m_Validate = True Then  'schema must be specified

    Try

      'get instance of validator from separate function
      Dim objValidator As XmlValidatingReader = GetValidatingReader()

      'add the event handler for any validation errors found
      AddHandler objValidator.ValidationEventHandler, _
                 AddressOf ValidationError

      Return objValidator   'return it to the calling routine

    Catch objErr As Exception

      Throw objErr            'throw exception to calling routine

    End Try

  Else   'no schema provided

    'create a new Exception and fill in details
    Dim objSchemaErr As New Exception("You must provide a Schema" _
```

```
                    & " when using the GetSuppliersValidatingReader method")
      objSchemaErr.Source = "SupplierListXml"

      Throw objSchemaErr  'throw exception to calling routine

    End If

  End Function
```

You can also see here how we create a custom exception to return to the calling routine if the path of the XML schema was not provided when the component's constructor was originally called. Obviously, we can't perform validation unless we have a schema. It is possible to use an inline schema for validation (one that is part of the XML document), but we haven't enabled that in our application.

> *In general, when using a "standard" XML document format for data transport on a regular basis (in other words a document that is of the same structure each time), using an inline schema only adds to the payload sent across the network. It's probably better in this situation for each party to have a copy of the schema locally and validate against that.*

Creating and Initializing an XmlValidatingReader Object

The `GetValidatingReader` function we use in our `GetSuppliersValidatingReader` method is implemented by a separate routine, which is listed next. We simply create an `XmlTextReader` for the document (in our case a disk file), then use this in the constructor for an `XmlValidatingReader`. Next we set the validation type (in our case we use `Auto`, so that it will detect the schema type automatically – it can handle XSD and XDR schemas and DTDs):

```
Private Function GetValidatingReader() As XmlValidatingReader

  'create new XmlTextReader object and load XML document
  m_XTReader = New XmlTextReader(m_XmlFilePath)

  'create an XmlValidatingReader for this XmlTextReader
  Dim objValidator As New XmlValidatingReader(m_XTReader)

  'set the validation type to "Auto"
  objValidator.ValidationType = ValidationType.Auto
```

We also need to create an `XmlSchemaCollection` object that references the schema(s) we want to validate against (there could be more than one if they inherit from each other), add our schema to it, and assign this collection to the `Schemas` property of the `XmlValidatingReader` before returning it to the calling routine. The version (overload) of the `Add` method we are using to add our schema to the `XmlSchemaCollection` takes two string parameters. This first is the namespace for the schema (we use the default here by providing an empty string for this parameter) and the second is the path to the schema file:

```
  'create a new XmlSchemaCollection
  Dim objSchemaCol As New XmlSchemaCollection()

  'add our schema to it
  objSchemaCol.Add("", m_SchemaPath)
```

```
    'assign the schema collection to the XmlValidatingReader
    objValidator.Schemas.Add(objSchemaCol)

    Return objValidator  'return to calling routine

  End Function
```

Catching Validation Events

If we just use the `XmlValidatingReader` as it is now, any validation errors that are detected while reading the XML document will cause an exception. Often we want to handle these validation errors separately, or at least get specific information about the error. We can do this by handling the event that the `XmlValidatingReader` raises when a validation error is encountered.

If you look back at the definition of the `GetSuppliersValidatingReader` method, you'll see that we assigned an event handler within our data access component to this event:

```
'add the event handler for any validation errors found
AddHandler objValidator.ValidationEventHandler, _
           AddressOf ValidationError
```

The event handler is a separate subroutine that accepts an instance of a `ValidationEventArgs` object as the second parameter. This object contains several details of the error, although we are only using the `Message` property in our example. We just create a custom exception, and raise it to the calling routine:

```
    Private Sub ValidationError(ByVal objSender As Object, _
                                ByVal objArgs As ValidationEventArgs)

      'create a new Exception and fill in details
      Dim objValidErr As New Exception("Validation error: " _
                                       & objArgs.Message)
      objValidErr.Source = "SupplierListXml"

      Throw objValidErr  'throw exception to calling routine

    End Sub
```

Displaying the Results of the GetSuppliersValidatingReader Method

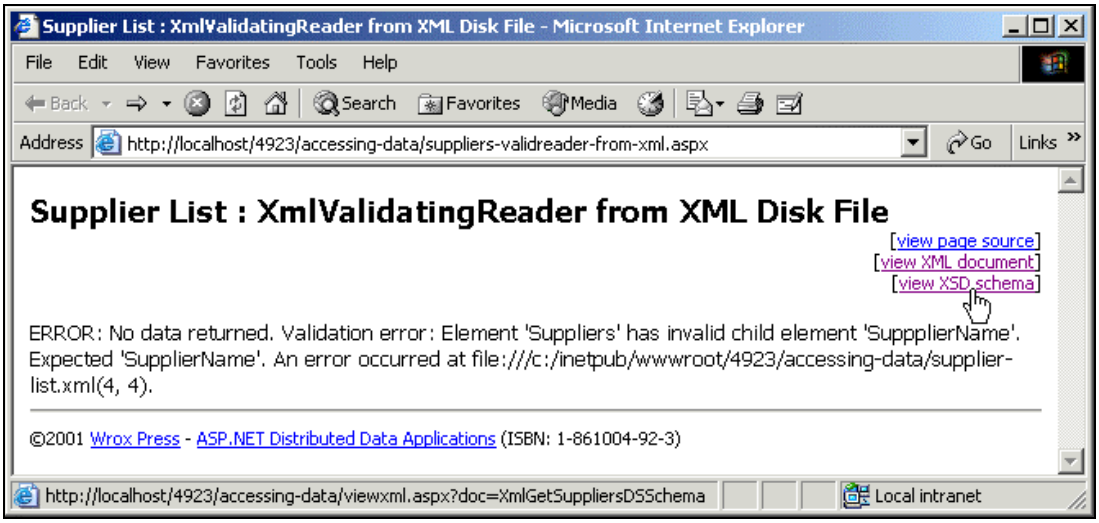
Our example page, `suppliers-validreader-from-xml.aspx`, uses the `GetSuppliersValidatingReader` method, and displays the results of reading the disk file named `supplier-list.xml` and validating it against an XML schema named `supplier-list.xsd`. The code in the `Page_Load` event creates the physical path to the document and the schema files, and then creates an instance of the data access component (specifying both of these paths):

```
'create physical path to XML file (in same folder as ASPX page)
Dim strCurrentPath As String = Request.PhysicalPath
Dim strPathOnly As String = Left(strCurrentPath, _
                                 InStrRev(strCurrentPath, "\"))
Dim strXMLPath As String = strPathOnly & "supplier-list.xml"
Dim strSchemaPath As String = strPathOnly & "supplier-list.xsd"

'create an instance of the data access component
Dim objSupplierList As New Wrox4923Xml.SupplierListXml(strXMLPath, _
                                                       strSchemaPath)
```

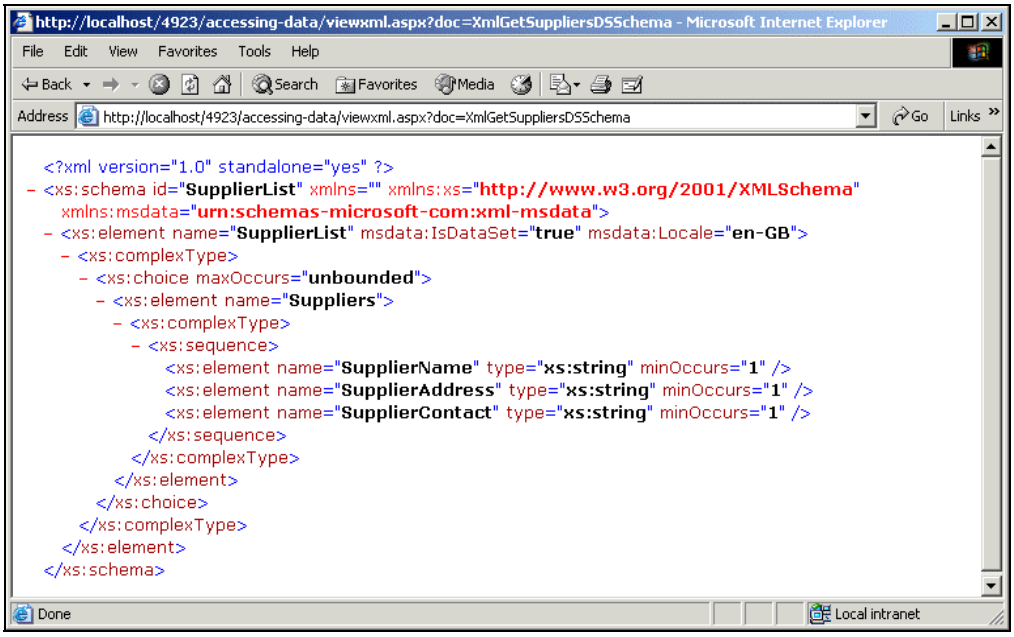
Then it calls the `GetSuppliersValidatingReader` method to get back the `XmlValidatingReader`, and displays the results using the same `ParseXmlContent` method as we did in the previous example:

```
'declare variable to hold XmlValidatingReader object
Dim objReader As XmlValidatingReader

'call the method to return the data as an XmlValidatingReader
objReader = objSupplierList.GetSuppliersValidatingReader()

'if we got a result, display the parsed content
If Not objReader Is Nothing Then
  lblMessage.Text = ParseXmlContent(objReader)
End If

objReader.Reader.Close()   'close the Reader to release file
```

As we use the same XML document and the same `ParseXmlContent` method as the previous example, the output is identical to the screenshot we showed in that example. To see a validation error, all you have to do is edit the XML source file named `supplier-list.xml`, for example here is the result after we changed the name of one of the `<SupplierName>` elements:



You can also view the schema we're validating against by clicking the links at the top right of the page:



Returning a DataSet Object

So far we've returned our XML document through a couple of objects that are part of the XML armory – defined within `System.Xml` and it's subordinate namespaces. However, we can take advantage of the synchronization between XML and relational data formats that we discussed in the previous chapter to return our XML document using objects that are normally associated with the relational data world.

The next example demonstrates this by returning an ADO.NET `DataSet` object containing a relational representation of our XML document:

```
Public Function GetSuppliersDataSet() As DataSet
```

To "convert" XML into a relational format, we can use an `XmlDataDocument` object. In Chapter 1 we discussed how this maintains and exposes two synchronized "views" of the data it contains. It exposes the normal XML DOM and XPath-compatible view of the data in the same way as is exposed by an `XmlDocument` object. It also has a `DataSet` property that returns an ADO.NET `DataSet`-compatible view of the data.

So, all we need to do is load our XML document into an `XmlDataDocument` object, and return a reference to its `DataSet` property. However, to be able to access this, we have to use a schema as well, and load this into the `XmlDataDocument` object before we load the XML document itself.

In our example, we also take advantage of the `GetValidatingReader` function within our data access component to read the XML document and validate it as we load it into the `XmlDataDocument`. So, the first step is to get the `XmlValidatingReader` that references the XML document, and we also have to specify the event handler we want to use for validating it (as in our previous example):

```
    'get instance of validator from separate function
    Dim objValidator As XmlValidatingReader = GetValidatingReader()

    'add the event handler for any validation errors found
    AddHandler objValidator.ValidationEventHandler, AddressOf ValidationError
```

Now we can create the new `XmlDataDocument` object, load the schema from disk, then load the XML document via our `XmlValidatingReader`. Once loaded, we close the `XmlTextReader` from which we created the `XmlValidatingReader` (this is why it's declared as a member variable within our component), and return the `DataSet` reference:

```
    'create a new XmlDataDocument object
    Dim objDataDoc As New XmlDataDocument()

    'load schema into DataSet exposed by XmlDataDocument object
    objDataDoc.DataSet.ReadXmlSchema(m_SchemaPath)

    'load document - its validated against Schema as it loads
    objDataDoc.Load(objValidator)

    'close the XmlTextReader
     m_XTReader.Close()

    'return the DataSet to the calling routine
    Return objDataDoc.DataSet
```

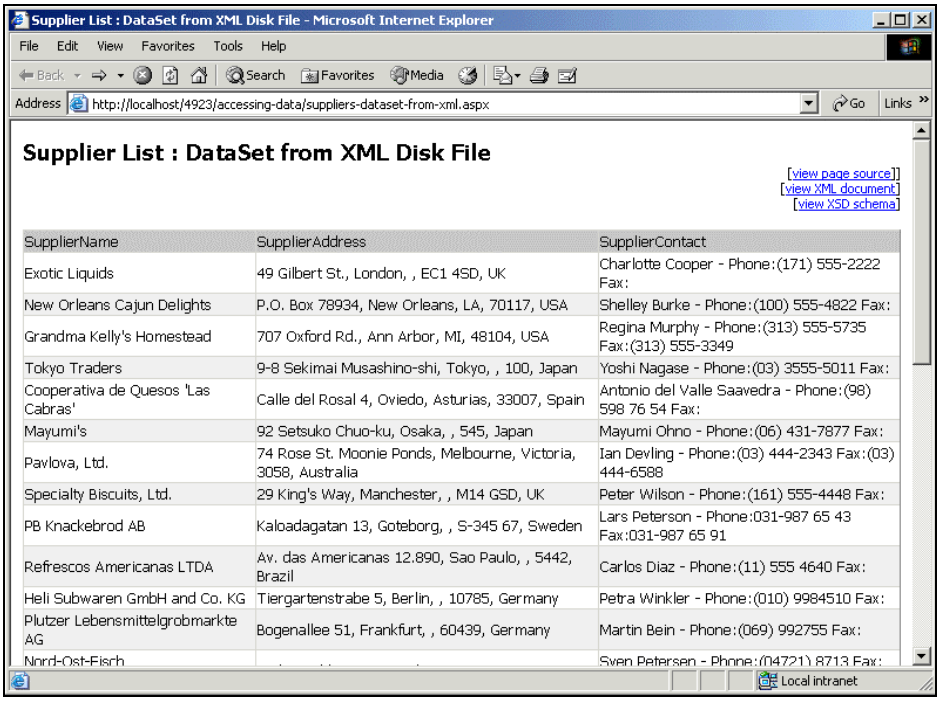
We've removed the error-handling code from this listing to make it easier to follow, though it is in the examples you can download and run on your own system.

Displaying the Results

The example page `suppliers-dataset-from-xml.aspx` uses the `GetSuppliersDataSet` method and displays the results. As we get back a `DataSet` object, we can display the rows in it using a `DataGrid` control like we did in many examples in the previous chapter (which also helps to prove that it is a real `DataSet` we get back!):

```
<asp:DataGrid id="dgrSuppliers" runat="server">
<HeaderStyle BackColor="#c0c0c0"></HeaderStyle>
<AlternatingItemStyle BackColor="#eeeeee"></AlternatingItemStyle>
</asp:DataGrid>
```

This is what the page looks like when you open it in the browser. There are the now-customary links to show the source code, the XML document and the schema we used:



In the `Page_Load` event, all we need to do is create the physical paths to the XML document and the schema we want to use, and create our data access component instance (as in the previous examples). Then we call the `GetSuppliersDataSet` method and assign the result to the `DataGrid` control. As a `DataSet` can contain more than one table, we also have to specify the name of the table that we want to act as the data source for the grid, and finally call the `DataBind` method:

```
'create an instance of the data access component
Dim objSupplierList As New Wrox4923Xml.SupplierListXml(strXMLPath, _
                                                       strSchemaPath)

'call the method to return the data as a DataSet and
'assign it to the DataGrid server control for display
dgrSuppliers.DataSource = objSupplierList.GetSuppliersDataSet()

'check we got a result - will be Nothing if there was an error
If Not dgrSuppliers.DataSource Is Nothing Then

  'set data member and bind the data to display it
  dgrSuppliers.DataMember = "Suppliers"
  dgrSuppliers.DataBind()

End If
```

Returning a Custom Array Reference

One of the examples we used with our relational data source in the previous chapter returned the supplier list as an array of strings. We can quite easily do the same from an XML document if this is the format that we need for our application. The method named `GetSuppliersArray` returns the same set of values as the relational example from the previous chapter. Of course, the way it is implemented within our XML data access component is quite different.

In this method, we use an `XPathDocument` to hold our XML while we extract the values we want from it. An `XPathDocument` object is a special version of the "document" objects provided in the `System.Xml` (and its child) namespaces. It is designed to provide fast and efficient access to the document contents using "path" definitions from the W3C XPath query language. It does not support the XML DOM techniques for accessing the content, and is consequently "lighter" and more efficient if we only need to use XPath queries.

One of the methods within our data access component, `GetSuppliersXPathDocument`, returns the data as an `XPathDocument` to the calling routine. We use it in a later example in this chapter, and we'll look at it in more detail there. In the meantime, we can use this method from within our `GetSuppliersArray` method to get an instance of `XPathDocument` that contains the XML stored in the disk file.

```
Public Function GetSuppliersArray() As Array

  'use function in this class to get an XPathDocument
  Dim objXPathDoc As XPathDocument = GetSuppliersXPathDocument()
```

To access an `XPathDocument`, we can use XPath queries directly. However, a more generally useful technique when we want to iterate through the document content is to create an `XPathNavigator` object based on the `XPathDocument` object:

```
  'create a new XPathNavigator object using the XPathDocument object
  Dim objXPNav As XPathNavigator = objXPathDoc.CreateNavigator()
```

Using the XPathNavigator and XPathNodeIterator Objects

Now we can use the `XPathNavigator` to access the contents of our XML document. We declare a couple of variables to hold `XPathNodeIterator` objects. These are collection-like objects designed to hold a set of nodes from an XML document and they support iterating through the nodes contained, as we'll see shortly.

```
  'declare variables to hold two XPathNodeIterator objects
  Dim objXPRowIter, objXPColIter As XPathNodeIterator
```

We move to the first child element in our document, which will be the root element in this case as we created the `XPathNavigator` on the document object itself (we could have created it pointing to a specific node if required by calling the `CreateNavigator` method of that node instead). Then we use the `Select` method of the `XPathNavigator` to return an `XPathNodeIterator` that references all the nodes that match a specified XPath expression. In our case, the XPath expression is "`child::*`", meaning all child nodes (the `<Suppliers>` nodes in the XML document):

```
'move to document element
  objXPNav.MoveToFirstChild()

  'select all the child nodes of the document node into an
  'XPathNodeIterator object using an XPath expression
  objXPRowIter = objXPNav.Select("child::*")
```

We can find out how many nodes we got from the `Count` property of the `XPathNodeIterator`, and store this away in a variable. Then we move to the first child of this node (the `<SupplierName>` element in our document) and repeat the process to find out how many child nodes this node has:

```
  'get number of "rows" (number of child elements)
  Dim intLastRowIndex As Integer = objXPRowIter.Count - 1

  'move to first child of first "row" element
  objXPNav.MoveToFirstChild()

  'select all the child nodes of this node into another
  'XPathNodeIterator object using an XPath expression
  objXPColIter = objXPNav.Select("child::*")

  'get number of "columns" (one per child element)
  Dim intLastColIndex As Integer = objXPColIter.Count - 1
```

Building the Array

We now know how "long" our array needs to be (the number of "rows" in the data), and how many values there are for each "row" (the number of "columns" in the data). Hence we can declare a suitably sized array:

```
  'can now create an Array of the appropriate size
  Dim arrResult(intLastColIndex, intLastRowIndex) As String
  Dim intLoop As Integer   'to hold index into array
```

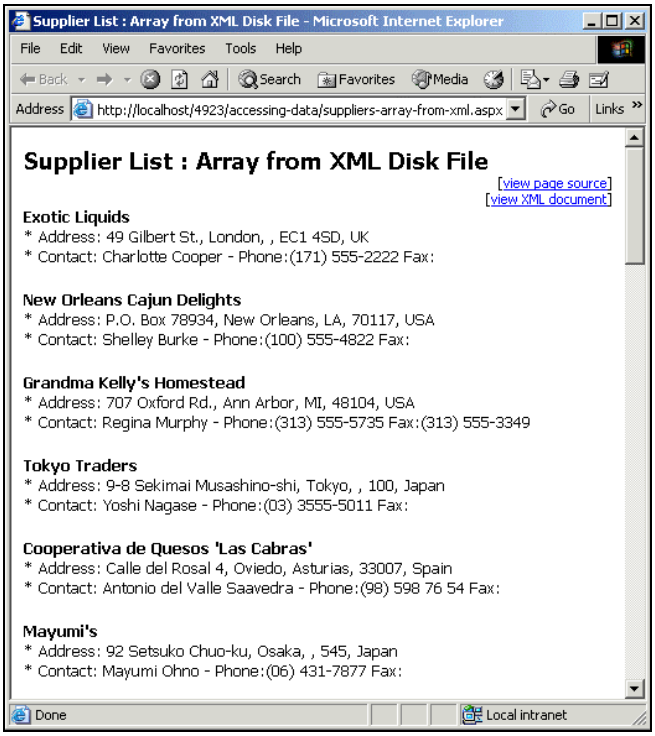
At last we're in a position to collect the data from the XML into our new array. We iterate through the "rows" of data using the `XPathNodeIterator` we set to the collection of `<Supplier>` elements. For each one, we can iterate through the child elements (the `<SupplierName>`, `<SupplierAddress>` and `<SupplierContact>` elements) and fill in our array as we go. Afterwards, we just return the array to the calling routine:

```
  'iterate through the "rows"
  While objXPRowIter.MoveNext

    'create an XPathNavigator for this "row" element
    objXPNav = objXPRowIter.Current

    'get an XPathNodeIterator containing the child nodes
    objXPColIter = objXPNav.Select("child::*")
```

```
      'iterate through these child nodes adding values to array
      For intLoop = 0 To intLastColIndex
        objXPColIter.MoveNext()
        arrResult(intLoop, objXPRowIter.CurrentPosition - 1) _
                                  = objXPColIter.Current.Value
      Next

    End While

    'return the array to the calling routine
    Return arrResult

  End Function
```

Displaying the Results

The result of calling this method is a simple two-dimensional array of strings, which is exactly the same as the one we got back from the `GetSuppliersArray` method of our relational data access component in the previous chapter. The version of the page that uses our XML data source, `suppliers-array-from-xml.aspx`, is therefore just about identical to the "relational" version, both in the way it works and the output it produces:



In fact, the only difference is that we have to create the physical path to the XML document disk file we want to use, as in our previous examples in this chapter. Also, of course, we need to create an instance of the XML data access component this time. We haven't listed the code, but you can view it using the link at the top of the page.

Returning an ArrayList Reference

The second type of collection object we demonstrated in the previous chapter was the `ArrayList`, We also provide a method in our XML data access component that returns the data from the XML disk file as an `ArrayList`.

An `ArrayList` can only hold one "value" in each row. This means that the code is actually a lot simpler than the previous example in this chapter, which returned a two-dimensional string array. Our method accepts a parameter that we set to the element name (the "column" of the data) that we want to return in our `ArrayList`.

In our example page (coming later) you'll see that we specified the `SupplierName` element, and so our `ArrayList` will just contain the supplier names (as in the previous chapter `ArrayList` example). In our method, we use the same approach as the previous example, creating an `XPathDocument` and an `XPathNavigator` based on it. We also create a single `XPathNodeIterator` object:

```
Public Function GetSuppliersArrayList(ByVal strElementName As String) _
                                                          As ArrayList

  'use function in this class to get an XPathDocument
  Dim objXPathDoc As XPathDocument = GetSuppliersXPathDocument()

  'create a new XPathNavigator object using the XPathDocument object
  Dim objXPNav As XPathNavigator = objXPathDoc.CreateNavigator()

  'declare variable to hold an XPathNodeIterator object
  Dim objXPIter As XPathNodeIterator
```

However, now we can just move to the document element and use an XPath query that selects all the required element nodes in the document. We use the XPath predicate `descendant` rather than `child` this time, as the nodes we want aren't actually direct children of the root node – they are one level deeper than this:

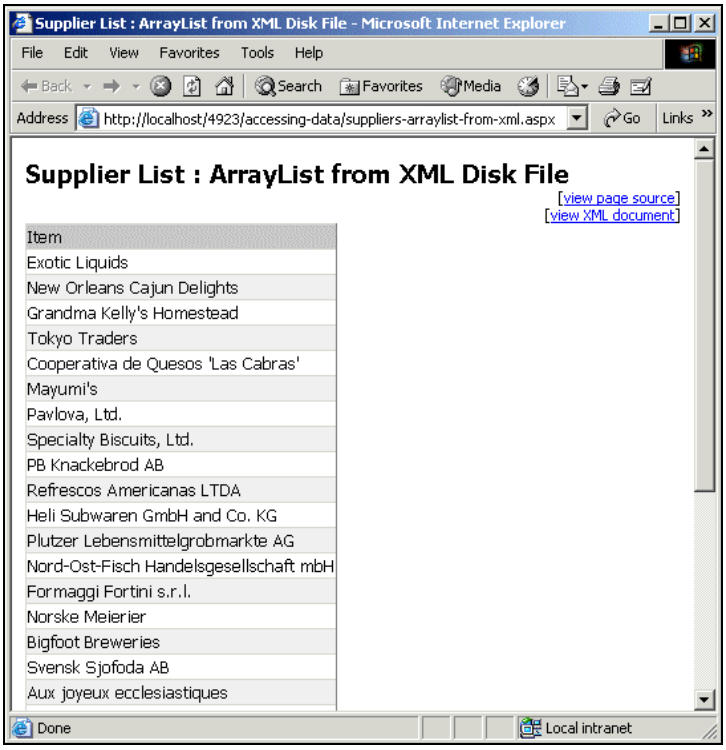
```
  'move to document element
  objXPNav.MoveToFirstChild()

  'select the required descendant nodes of document node into
  'an XPathNodeIterator object using an XPath expression
  objXPIter = objXPNav.Select("descendant::" & strElementName)
```

From here, we can now create an `ArrayList` and iterate through the collection of elements, copying their values into the `ArrayList`. We finish up by passing the `ArrayList` back to the calling routine:

```
  'create an ArrayList to hold the results
  Dim arrResult As New ArrayList()

  'iterate through the element nodes in the XPathNodeIterator
  'collection adding their values to the ArrayList
  While objXPIter.MoveNext()
    arrResult.Add(objXPIter.Current.Value)
  End While

  Return arrResult   'and return it to the calling routine

End Function
```

Displaying the Results

You won't be the least surprised to know that the page we use to display the contents of the `ArrayList` returned by our data access component in this example is just about identical to the version we used with an `ArrayList` in the previous chapter. The page `suppliers-arraylist-from-xml.aspx` is shown in the next screenshot:



It uses an ASP `DataGrid` control bound to the `ArrayList` to display the contents:

```
<asp:DataGrid id="dgrSuppliers" runat="server">
<HeaderStyle BackColor="#c0c0c0"></HeaderStyle>
<AlternatingItemStyle BackColor="#eeeeee"></AlternatingItemStyle>
</asp:DataGrid>
```

In the `Page_Load` event, we create the physical path to the XML document disk file (as in our previous examples in this chapter). We then create an instance of the XML data access component, and call the `GetSuppliersArrayList` method. Notice that this time we specify the `SupplierName` element as the one we want to select the values from:

```
    dgrSuppliers.DataSource = _
                objSupplierList.GetSuppliersArrayList("SupplierName")
```

We assign the result of the method to the `DataGrid` (as shown above) and then call `DataBind` to display the data. Again, we haven't listed all the code here, but you can view it using the link at the top of the page.

Returning an XML String

The final three methods in our XML data access component show how we can return an XML document in three "document" formats – a `String`, an `XmlDocument` object and an `XPathDocument` object. If we are simply streaming the XML to a non-.NET client or object, we will usually choose to return a string containing the XML. This can be sent to a client as the HTTP response across the network.

However, if we are dealing with a .NET client or object, we might prefer to return an `XmlDocument` or `XPathDocument` object that they can manipulate directly. Chapter 4, which introduces the .NET Remoting technology, demonstrates how we can send objects like this across an HTTP network such as the Internet.

Therefore, the format we choose for an application depends on how we intend to remote the data to the client or middle tier, and what the client or middle-tier objects will be doing with the data. We'll start by showing you how we can return an XML string from a component, and the following listing of the `GetSuppliersXmlString` in our example data access component does just that:

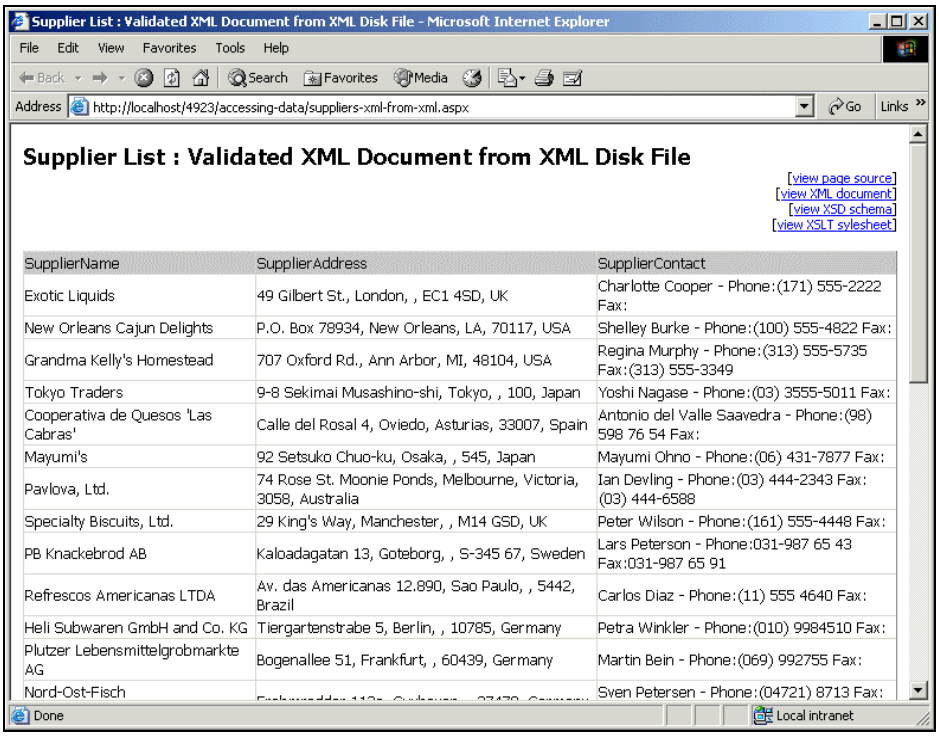
```
Public Function GetSuppliersXmlString() As String

  Try

    'use function in this class to get an XmlValidatingReader
    Dim objValidator As XmlValidatingReader = GetSuppliersValidatingReader()

    'create a new XmlDocument to hold the XML as it is validated
    Dim objXmlDoc As New XmlDocument()
    objXmlDoc.Load(objValidator)

    'return the complete XML content of the document
    Return objXmlDoc.OuterXml

  Catch objErr As Exception

    Throw objErr          'throw exception to calling routine

  Finally

    m_XTReader.Close()    'close the XmlTextReader

  End Try

End Function
```

You can see that we're validating the XML in this example by calling the method within our component named `GetSuppliersValidatingReader`, which we saw earlier in this chapter. It returns a reference to an initialized `XmlValidatingReader` for the document, so we can load this into a new `XmlDocument` object as it stands. The `XmlDocument` object will automatically call the `Read` method of the underlying `XmlTextReader` to get the contents of the document, and any validation errors will be flagged up as this process proceeds.

After remembering to close the `XmlTextReader`, we can then return the `OuterXml` property of the `XmlDocument` object. This is one of the useful extensions to the XML DOM that Microsoft implements within the `XmlDocument` object. It saves us having to iterate through the object ourselves extracting each element in turn. Other similar and useful extension properties are `InnerXml`, which includes all the XML and text content but without the containing element tags, and `InnerText`, which includes only the text content of the node and all its descendants.

Displaying the Results

The example page `suppliers-xml-from-xml.aspx` displays the XML string that our data access component generates. As usual, you can view the source of the page, the XML string itself, the schema we used to validate it and the XSLT stylesheet through the links at the top of the page:



This page uses the same technique as the examples in the previous chapter that display XML documents. We declare an `asp:Xml` server control in the page:

```
<asp:Xml id="xmlResult" runat="server" /><br />
```

In the `Page_Load` event, we create the physical paths to the XML document and the stylesheet, and create our data access component instance, exactly as in the previous examples in this chapter. Then we can call the `GetSuppliersXmlString` method of our component, and assign the result to the `DocumentContent` property of the `asp:Xml` control. We also assign the path to our XSLT stylesheet to the `TransformSource` property of the control:

```
'create an instance of the data access component
Dim objSupplierList As New Wrox4923Xml.SupplierListXml(strXMLPath, _
                                                       strSchemaPath)

'call the method to return the data as an Xml String and
'assign it to the Xml Server Control
xmlResult.DocumentContent = objSupplierList.GetSuppliersXmlString()

'specify path to XSLT stylesheet that transforms XML for display
xmlResult.TransformSource = "supplier-list-style.xsl"
```

Returning an XmlDocument Object

If we need an `XmlDocument` object, rather than just a string that contains the document, we can use the `GetSuppliersXmlDocument` method of our data access component. In this case, we can instantiate the component with or without specifying an XML schema. The function first creates a new empty `XmlDocument` object, and then checks the member variable `m_Validate` to see if a schema was specified in the constructor to the component when it was instantiated:

```
Public Function GetSuppliersXmlDocument() As XmlDocument

  'create a new XmlDocument object
  Dim objXmlDoc As New XmlDocument()

  If m_Validate = True Then  'validate against schema

    Try

      'use function in this class to get an XmlValidatingReader
      Dim objValidator As XmlValidatingReader = _
                                          GetSuppliersValidatingReader()

      'load the XML and validate as it's being loaded
      objXmlDoc.Load(objValidator)

    Catch objErr As Exception

      Throw objErr    'throw exception to calling routine

    Finally

      m_XTReader.Close()   'close the XmlTextReader

    End Try

  Else  'validation not required
    ....
```

If there was a schema specified, it calls the method in our component we saw earlier to get an `XmlValidatingReader` that references the XML disk file, and uses this to load the XML into the `XmlDocument` object (in the code above). Any error that is detected when loading the document is raised to the calling routine.

Alternatively, if there was no schema specified, we simply use the `Load` method of the `XmlDocument` object we created at the beginning of the method to load the XML from disk directly (in the following code). Again, any error is raised to the calling routine:

```
    ....

    Try

      'load the XML from disk without validation
      objXmlDoc.Load(m_XmlFilePath)

    Catch objErr As Exception

      Throw objErr    'throw exception to calling routine

    End Try

  End If

  'return the XmlDocument object
  Return objXmlDoc

End Function
```

Displaying the Results

The example page that uses this method, `suppliers-xmldoc-from-xml.aspx`, is identical to the previous example except that it now assigns the result to the `Document` property of the `asp:Xml` control. This property expects an `XmlDocument` object, whereas the `DocumentSource` property expects a string. We also specify the same XSLT stylesheet, which creates the table in the page:

```
    'call the method to return the data as an XmlDocument and
    'assign it to the Xml Server Control
    xmlResult.Document = objSupplierList.GetSuppliersXmlDocument()

    'specify path to XSLT stylesheet that transforms XML for display
    xmlResult.TransformSource = "supplier-list-style.xsl"
```

Returning an XPathDocument Object

The final example in this chapter demonstrates how we can return an `XPathDocument` object from an XML disk file. In fact, we used this method a couple of times earlier on within other methods that we discussed. We'll see how it works here.

Fundamentally, the technique is similar to that used in the previous example to create and return an `XmlDocument` object. However, an `XPathDocument` object does not have a `Load` method, so we can't create an empty one and load the XML afterwards like we did with an `XmlDocument` object. Instead, we have to specify the source of the XML document in the constructor for the `XPathDocument`.

So, our method first checks to see if a schema was specified when the component's constructor was called. If so, it uses the `GetSuppliersValidatingReader` method in our component to get an `XmlValidatingReader` that references the XML disk file. Then it calls the constructor for a new `XPathDocument` object, specifying the `XmlValidatingReader` as the source of the document:

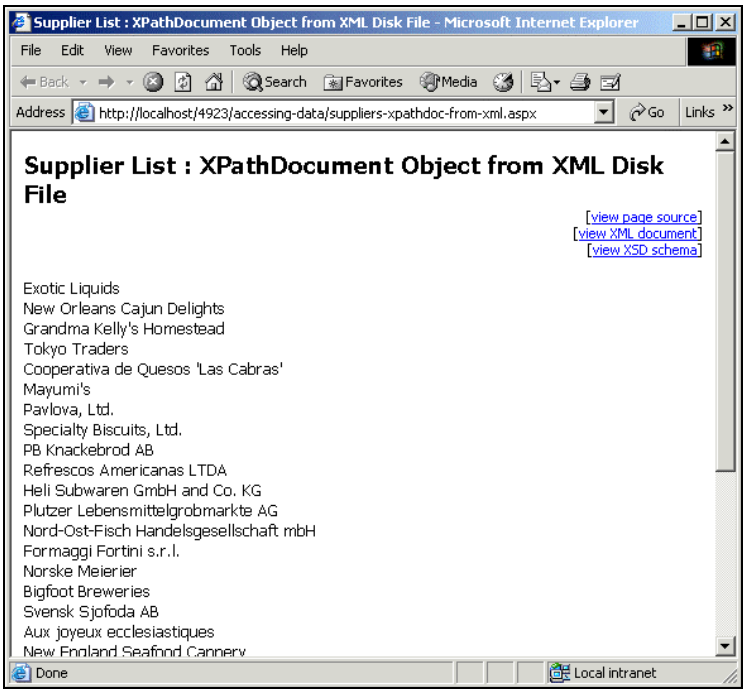
```
Public Function GetSuppliersXPathDocument() As XPathDocument

  'declare a variable to hold an XPathDocument object
  'cannot create an "empty" one and load the XML afterwards
  Dim objXPathDoc As XPathDocument

  If m_Validate = True Then  'validate against schema

    Try

      'use function in this class to get an XmlValidatingReader
      Dim objValidator As XmlValidatingReader = _
                          GetSuppliersValidatingReader()

      'load the XML and validate as it's being loaded
      objXPathDoc = New XPathDocument(objValidator)

    Catch objErr As Exception

      Throw objErr    'throw exception to calling routine

    Finally

      m_XTReader.Close()   'close the XmlTextReader

    End Try

  Else  'validation not required
    ...
```

If validation is not required, in other words no schema was specified in the constructor for the component, we simply create the `XPathDocument` by specifying the physical path to the XML disk file instead:

```
    ...

    Try

      'load the XML from disk without validation
      objXPathDoc = New XPathDocument(m_XmlFilePath)

    Catch objErr As Exception

      Throw objErr    'throw exception to calling routine

    End Try

  End If

  Return objXPathDoc   'return it to the calling routine

End Function
```

Displaying the Results

The example page that uses the `GetSuppliersXPathDocument` method is a little different from our earlier examples that displayed an XML document. This page, `suppliers-xpathdoc-from-xml.aspx`, displays just the supplier names as a list:



To do this, we use an `asp:Label` control declared in the page:

```
<asp:Label id="lblMessage" runat="server" />
```

In the `Page_Load` event we still create the physical paths to our XML document and XML schema disk files, and create our data access component instance as before. However, we also declare a variable to hold the `XPathDocument` that our method returns so that we can work with it to extract the data we want to display (we've removed some of the error-handling code for clarity here):

```
'declare a variable to hold an XPathDocument object
Dim objXPDoc As XPathDocument

'create an instance of the data access component
Dim objSupplierList As New Wrox4923Xml.SupplierListXml(strXMLPath, _
                                                       strSchemaPath)

'call the method to return the data as an XPathDocument
objXPDoc = objSupplierList.GetSuppliersXPathDocument()
```

Now we create an `XPathNavigator` based on the `XPathDocument`, and a variable to hold an `XPathNodeIterator` object like we used in our `Array` and `ArrayList` examples. We move the `XPathNavigator` to the root element of the document using the `MoveToFirstChild` method then create a collection in the `XPathNodeIterator` of all the `SupplierName` elements:

```
'create an XPathNavigator object against the XPathDocument
Dim objXPNav As XPathNavigator = objXPDoc.CreateNavigator()

'declare variable to hold an XPathNodeIterator object
Dim objXPIter As XPathNodeIterator

'move to document element
objXPNav.MoveToFirstChild()

'select all the SupplierName nodes of the document node into an
'XPathNodeIterator object using an XPath expression
objXPIter = objXPNav.Select("descendant::SupplierName")
```

Then it's simply a matter of iterating through the collection and building up a string to display in the `asp:Label` control on our page:

```
Dim strResult As String   'to hold results for display

'iterate through the element nodes in the XPathNodeIterator
'collection adding their values to the output string
While objXPIter.MoveNext()
  strResult += objXPIter.Current.Value & "<br />"
End While

lblMessage.Text = strResult
```

Summary

We've spent a lot of time in this and the previous chapter looking at various types of data access component that the .NET Framework makes so easy to build. The foundation for any distributed (or, for that matter, non-distributed) data application is its ability to fetch and update data in a way that is efficient, but also maintainable, extensible and flexible.

Having to completely rebuild an application just because the data source has changed is not a great approach for developers in today's results-oriented world. By separating out the data access features into a separate physical or logical "tier", we allow the application to evolve as the data source changes, and adapt to new requirements more easily.

What we've attempted to demonstrate in these two chapters is the various ways that we can access data in different formats, and yet expose it to our applications in the format that best suits them. We aren't shackled with just a **relational view**, or just an **XML view** as we often were in the past. As you've seen, we can freely intermix these to get the output format we want from almost any data source. On top of that, we can choose between a **connected** approach (such as the `DataReader` or `XmlReader` objects), or a **disconnected** approach (using a `DataSet` or an XML "document-style" object).

The topics we covered in this chapter were:

- How we can build components that access XML documents
- How we can convert and return the XML as various different object types
- How we can display the XML easily in our ASP.NET pages

So far, we've only concerned ourselves with **reading** data. In later chapters, we'll return to building data access components to see how we can go about **updating** the source data from a component, and returning information about the success or otherwise of these updates. We'll also see how we can use components like the ones we saw in this chapter within some example applications.