Database » Database » ADO.NET

VB, Windows, .NET, Visual Studio, ASP.NET, ADO, ADO.NET, Dev

.NET DataSet to ADODB Recordset Conversion

By KRucker

Posted : 8 Mar 2006
Updated : 8 Mar 2006
Views : 42,307
Bookmarked : 17 times

A description of how how to convert a .NET DataSet to an ADODB Recordset.

13 votes for this Article.
Popularity: 4.81 Rating: **4.32** out of 5
1 2 3 4 5

Download source files - 2.44 Kb

Contents

Introduction

I was tasked with implementing a .NET Web Service to provide access to our application data by outside entities. The Web Service must support clients written in .NET languages, Visual Basic 6.0, and non-Microsoft languages. All access to our database must be through stored procedures, so we did not need to support running direct SQL queries. The design was to use a single method that accepts an XML string that describes the stored procedure to run, its parameters, and how to return the data if any data is returned (`DataSet` XML or `Recordset` XML). The implementation used the .NET data access objects to return a `DataSet` and ADODB objects to return a `Recordset`. The returned data was then serialized to an XML string and returned to the caller. At first, this worked like a charm, however, as soon as the load testing started, it was obvious, I had a problem. At the web server, a single user returning a large set of data utilized 70% of the CPU; two simultaneous users pegged it at 100%, five or more users - some would get timeouts! This was very bad!

I knew going in that there would be some performance penalty for using ADODB in the .NET code, but I never dreamed it would be so pronounced. It seems that the penalty for using COM interop from .NET can be quite severe. Back to the drawing board.

Now for the problem, I knew from the beginning that if I had to provide the XML node list that a `DataSet` persists in a form, along with the XML schema, any .NET or non-Microsoft client should easily be able to use to utilize the data, then the plan for returning persisted `Recordset` formatted XML to VB 6.0 clients was in serious jeopardy. So far as I knew, there was no native way to convert a `DataSet` to an ADODB `Recordset` in VB.NET. Knowing that someone, somewhere was bound to have had this problem before me, I began to search the internet to see

if anyone had come up with a solution that I could use. I found a couple of articles that had VB 6.0 code that use the .NET node list and schema to construct an ADODB `Recordset`, but they were client side solutions, I needed something that would work on the server. I found a Microsoft Support article: How To Convert an ADO.NET DataSet to ADO Recordset in Visual Basic .NET, that appeared to be exactly what I had been searching for.

I followed the instructions in the article, set up the code, and it worked! The only problem I had now was that the example code from Microsoft was writing the XML to a file and reading an XSL file. Since I need this to be a server solution, I needed to eliminate the file IO. No big deal, I altered the code to generate the XSL on the fly in a string (it was quite small), and used a `MemoryStream` to contain the XML instead of a file. I tried this out and it worked. Now I had to test it against some real world data. I set up some test code to pull data from our test database, and sent it through the conversion function to see what I get on the other side.

The first problem I saw was that the only fields getting a data type were integers and strings, for all other fields the data type was blank. This was causing an error when attempting to load the `Recordset` from the ADODB Stream. I tracked down where the code was determining the data type, and sure enough, integers and strings were all they were trapping for. I added several data types and a `Case Else` that set anything I was not specifically trapping, to type string. Thinking this should work, I tried another run and still got an error loading the XML into the `Recordset`. I commented out all of the fields returned in the stored procedure except one, ran again, and it worked. I repeated this process, un-commenting another field each time, trying to figure out what type of field I was having a problem with. It turned out to be date-time fields. After a bit of research, I found that in the `DataSet`, the date-time format included time zone information. For the `Recordset` to accept a date-time with this additional time zone information, the data type in the XML must be set to `dateTime.iso8601tz`. I made this change, attempted another run, and it worksed.

Now I wrapped the code in a class and integrated it into the Web Service and began unit testing. I ran through a couple of calls, then hit another problem - binary fields. This one took a while to research, but I found that binary fields in a `DataSet` are persisted as base64 encoded strings; the ADODB `Recordset` expects binary fields to be persisted as binary hex encoded strings. Since the Microsoft code for transforming the data portion of the `DataSet` is the part that uses the XSL transformation, there was no opportunity to re-encode the data in a binary field. First, I tried setting the data type in the XML to `bin.base64`. This allowed the `Recordset` to load the data without an error, but, the binary data ended up stored in the `Recordset` as a string field containing the base64 encoded string. In order to have the `Recordset` convert the field to binary upon loading, as it should, it must be encoded in binary hex, and the data type in the XML set to `bin.hex`. To solve this, I rewrote the transformation code to loop through the `DataSet`, and added the data to the XML using the `XmlTextWriter` just like the rest of the code does to add the header and schema information. This gave me the chance to detect binary fields and binary hex encode them.

Now I needed to find or write a function to perform the binary hex encoding. I couldn't find any ready-made code on the internet, but, I did find information on binary hex encoding. `BinaryHex` encoding simply takes each octet (byte) of the binary stream, divides it into two 4 bit nibbles, and places the hexadecimal character representing that nibbles value in the output string. For example, if you have 32 bits of binary data:

```
10010010111100011010110011010100
```

divide into octets (bytes):

```
10010010  11110001  10101100  11010100
```

divide the octets into 4 bit nibbles:

```
1001  0010  1111  0001  1010  1100  1101  0100
```

decimal values of nibbles:

```
9  2  15  1  10  12  13  4
```

hexadecimal values:

```
9  2  F  1  A  C  D  4
```

encoded string representing the original 32 bit binary value:

```
"92F1ACD4
```

Armed with the above information, it was simple to write a small function to binary hex encode a byte array and return the resultant string. I added this to the class, modified the data transformation code to detect binary fields, and applied this new function to the data. Now I have a conversion class that I can use. The only thing that you may need to update is the function that determines the data type to put in the XML. If you need a data type that I'm not trapping, returning it as a string is not sufficient, just add a `Case` statement for your data type to the function.

Now, on to the code!

Background

Requirements:

- Microsoft Windows 2003, Windows XP, Windows 2000, or Windows NT 4.0 Service Pack 6a
- Microsoft Data Access Components (MDAC) 2.6 or later
- Microsoft Visual Studio .NET

This article assumes that you are familiar with the following topics:

- Microsoft Visual Basic .NET syntax
- ADO.NET and earlier versions of ADO
- `DataSet` and ADO `Recordset` XML formats

Using the code

It should be noted here that I started with the code in the above mentioned Microsoft Support article. The code presented, however, is a substantial update of that code.

Note: You must call the `FillSchema` method of the `DataAdapter` to obtain the schema information with your `DataSet`. If you do not, all fields will be created as a string data type.

The `GetADORS` function provides the entry point and logic flow for the class. It also creates the `MemoryStream` and `XmlTextReader` used by the rest of the functions to build the output XML string.

```
Public Function GetADORS(ByVal DS As DataSet, _
                ByVal dbName As String) As String

    Try
        'Create a MemoryStream to contain the XML
        Dim mStream As New MemoryStream
        'Create an XmlWriter object, to write
        'the formatted XML to the MemoryStream
        Dim xWriter As New XmlTextWriter(mStream, Nothing)

        'Additional formatting for XML
        xWriter.Indentation = 8
        xWriter.Formatting = Formatting.Indented
        'call this Sub to write the ADONamespaces
        WriteADONamespaces(xWriter)
        'call this Sub to write the ADO Recordset Schema
        WriteSchemaElement(DS, dbName, xWriter)
        'Call this sub to transform
        'the data portion of the Dataset
        TransformData(DS, xWriter)
        'Flush all input to XmlWriter
        xWriter.Flush()

        'Prepare the return value
        mStream.Position = 0
        Dim Buffer As Array
        Buffer = Array.CreateInstance(GetType(Byte), mStream.Length)
        mStream.Read(Buffer, 0, mStream.Length)
        Dim TextConverter As New UTF8Encoding
        Return TextConverter.GetString(Buffer)

    Catch ex As Exception
        'Returns error message to the calling function.
        Err.Raise(100, ex.Source, ex.ToString)
    End Try

End Function
```

First, I've added two lines that indicate to the `XmlTextWriter` that I want the XML to be indented. The entire purpose of this is to make the output XML human readable. These lines can be omitted if you like. Having the output XML easily readable made debugging the class much easier. Next, `WriteADONamespaces` is called to add the `Recordset` schema to the output XML. `WriteSchemaElement` is then called to add the schema elements. `TransformData` is called to properly format the data and add it to the output XML. Finally, the contents of the `MemoryStream` are prepared for return as a string.

```
Private Sub WriteADONamespaces(ByRef xWriter As XmlTextWriter)
    'Uncomment the following line to change
    'the encoding if special characters are required
    'writer.WriteProcessingInstruction("xml",
    '      "version='1.0' encoding='ISO-8859-1'")

    'Add XML start element
```

```
    xWriter.WriteStartElement("", "xml", "")

    'Append the ADO Recordset namespaces
    xWriter.WriteAttributeString("xmlns", "s", Nothing, _
            "uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882")
    xWriter.WriteAttributeString("xmlns", "dt", Nothing, _
            "uuid:C2F41010-65B3-11d1-A29F-00AA00C14882")
    xWriter.WriteAttributeString("xmlns", "rs", Nothing, _
            "urn:schemas-microsoft-com:rowset")
    xWriter.WriteAttributeString("xmlns", "z", _
            Nothing, "#RowsetSchema")
    xWriter.Flush()
End Sub
```

The code in `WriteADONamespaces` is essentially unchanged from the code in the original Microsoft article. I have removed the comment describing the format of this section of the XML.

```
Private Sub WriteSchemaElement(ByVal DS As DataSet, _
        ByVal dbName As String, ByRef xWriter As _
        XmlTextWriter)

    'write element Schema
    xWriter.WriteStartElement("s", "Schema", _
            "uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882")
    xWriter.WriteAttributeString("id", "RowsetSchema")

    'write element ElementType
    xWriter.WriteStartElement("s", "ElementType", _
            "uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882")

    'write the attributes for ElementType
    xWriter.WriteAttributeString("name", "", "row")
    xWriter.WriteAttributeString("content", "", "eltOnly")
    xWriter.WriteAttributeString("rs", "updatable", _
            "urn:schemas-microsoft-com:rowset", "true")

    WriteSchema(DS, dbName, xWriter)
    'write the end element for ElementType
    xWriter.WriteFullEndElement()

    'write the end element for Schema
    xWriter.WriteFullEndElement()
    xWriter.Flush()
End Sub
```

The code in `WriteSchemaElement`, also, is essentially unchanged from the code in the original Microsoft article. I have removed the comment describing the format of this section of the XML.

```
Private Sub WriteSchema(ByVal DS As DataSet, ByVal dbName _
            As String, ByRef xWriter As XmlTextWriter)

    Dim i As Int32 = 1
    Dim DC As DataColumn

    For Each DC In DS.Tables(0).Columns

        DC.ColumnMapping = MappingType.Attribute

        xWriter.WriteStartElement("s", "AttributeType", _
                "uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882")
        'write all the attributes
```

```
            xWriter.WriteAttributeString("name", "", DC.ToString)
            xWriter.WriteAttributeString("rs", "number", _
                     "urn:schemas-microsoft-com:rowset", i.ToString)
            xWriter.WriteAttributeString("rs", "baseCatalog", _
                     "urn:schemas-microsoft-com:rowset", dbName)
            xWriter.WriteAttributeString("rs", "baseTable", _
                     "urn:schemas-microsoft-com:rowset", _
                     DC.Table.TableName.ToString)
            xWriter.WriteAttributeString("rs", "keycolumn", _
                     "urn:schemas-microsoft-com:rowset", _
                     DC.Unique.ToString)
            xWriter.WriteAttributeString("rs", "autoincrement", _
                     "urn:schemas-microsoft-com:rowset", _
                     DC.AutoIncrement.ToString)
            'write child element
            xWriter.WriteStartElement("s", "datatype", _
                    "uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882")
            'write attributes
            xWriter.WriteAttributeString("dt", "type", _
                     "uuid:C2F41010-65B3-11d1-A29F-00AA00C14882", _
                     GetDatatype(DC.DataType.ToString))
            xWriter.WriteAttributeString("dt", "maxlength", _
                     "uuid:C2F41010-65B3-11d1-A29F-00AA00C14882", _
                     DC.MaxLength.ToString)
            xWriter.WriteAttributeString("rs", "maybenull", _
                    "urn:schemas-microsoft-com:rowset", _
                     DC.AllowDBNull.ToString)
            'write end element for datatype
            xWriter.WriteEndElement()
            'end element for AttributeType
            xWriter.WriteEndElement()
            xWriter.Flush()
            i = i + 1
        Next
        DC = Nothing

    End Sub
```

The code in `WriteSchema`, also, is essentially unchanged from the code in the original Microsoft article. I have removed the comment describing the format of this section of the XML.

```
    Private Function GetDatatype(ByVal DType As String) As String
        Select Case (DType)
            Case "System.Int32", "System.Int16", "System.Integer"
                Return "int"
            Case "System.DateTime"
                Return "dateTime.iso8601tz"
            Case "System.String"
                Return "string"
            Case "System.Byte[]"
                Return "bin.hex"
            Case "System.Boolean"
                Return "boolean"
            Case "System.Guid"
                Return "guid"
            Case Else
                Return "string"
        End Select
    End Function
```

The `GetDatatype` function has been expanded to handle more data types than the original function. The original function only recognized `System.Int32` and `System.DateTime`. The

`Case Else` has also been added to return string type for all data types not in the `Case` statement.

```
Private Sub TransformData(ByVal DS As DataSet, _
            ByRef xWriter As XmlTextWriter)

    'Loop through DataSet and add data to XML
    xWriter.WriteStartElement("", "rs:data", "")
    Dim i As Long
    Dim j As Integer
    'For each row...
    For i = 0 To DS.Tables(0).Rows.Count - 1
        'Write the start element for the row
        xWriter.WriteStartElement("", "z:row", "")
        'For each field in the row...
        For j = 0 To DS.Tables(0).Columns.Count - 1
            'Write the attribute that describes
            'this field and it's value
            If DS.Tables(0).Columns(j).DataType.ToString_
                      = "System.Byte[]" Then
                'Binary data must be properly encoded (bin.hex)
                If Not IsDBNull(DS.Tables(0).Rows(i).Item(
                       DS.Tables(0).Columns(j).ColumnName)) Then
                    xWriter.WriteAttributeString(DS.Tables(0).
                       Columns(j).ColumnName, _
                       DataToBinHex(DS.Tables(0).Rows(i).Item(
                       DS.Tables(0).Columns(j).ColumnName)))
                End If
            Else
                If Not IsDBNull(DS.Tables(0).Rows(i).Item(
                       DS.Tables(0).Columns(j).ColumnName)) Then
                    xWriter.WriteAttributeString(
                             DS.Tables(0).Columns(j).ColumnName, _
           CType( _
              DS.Tables(0).Rows(i).Item(DS.Tables(0).
                          Columns(j).ColumnName), String))
                End If
            End If
        Next
        'End the row element
        xWriter.WriteEndElement()
    Next
    'Write the end element for rs:data
    xWriter.WriteEndElement()
    'Write the end element for xml
    xWriter.WriteEndElement()
    xWriter.Flush()

End Sub
```

`TransformData` adds the "rs:data" section of the XML. The function loops through the `DataSet` adding "z:row" elements for each data row. This function also adds the end tag for the root (XML) element.

```
Private Function DataToBinHex(ByVal thisData As Byte()) As String
    Dim sb As New StringBuilder
    Dim i As Integer = 0
    For i = 0 To thisData.Length - 1
        'First nibble of byte (4 most significant bits)
        sb.Append(Hex((thisData(i) And &HF0) / 2 ^ 4))
        'Second nibble of byte (4 least significant bits)
        sb.Append(Hex(thisData(i) And &HF))
```

```
        Next
        Return sb.ToString
    End Function
```

The `DataToBinHex` function performs the encoding of binary data.

Conclusion

As long as developers are faced with integrating .NET and VB 6.0, there will be a need to have the ability to pass data from one to the other. In the case where VB 6.0 is the client, the code presented here should help to alleviate the problem. While the code in the Microsoft article was a good starting point, it has a few shortcomings that prevent it from operating correctly in many real world situations. I believe that I have addressed the major concerns and shortfalls, and am presenting code that can be dropped into a project and used "as is" in most situations.

History

- Original submission - 03/08/2006.

License

This article has no explicit license attached to it but may contain usage terms in the article text or the download files themselves. If in doubt please contact the author via the discussion board below.

A list of licenses authors might use can be found here

About the Author

KRucker

Location: United States

Discussions and Feedback

15 messages have been posted for this article. Visit **http://www.codeproject.com/KB/database/ADOConversion.aspx** to post and view comments on this article, or click **here** to get a print view with messages.

PermaLink | Privacy | Terms of Use
Last Updated: 8 Mar 2006
Editor: Smitha Vijayan

Copyright 2006 by KRucker
Everything else Copyright © CodeProject, 1999-2008
Web12 | Advertise on the Code Project