

Doing the Impossible (Again)

William Vaughn

There are still a lot of Visual Basic 6.0 applications out there that call middle-tier objects that return ADO classic Recordsets. While ADO.NET can *read* a Recordset, it can't create one. This article shows how to use XML to create a Recordset from an existing DataTable.

We keep hearing about folks that have n-hundred-thousand line VB6 apps who want to convert/upgrade/port to VB.NET. The problem is that they're scared. Well, scared might not be accurate; perhaps "battle scarred" is a better term. In too many cases, every time they've tried upgrade in the past, they've been bit by the alligators they missed when they scouted the route. It's always the "unexpected" side-effects, issues, and bugs that drive us crazy and make us miss deadlines. One approach that seems to help is to migrate in phases. That is, convert the application a piece at a time. Sometimes (I'd like to think often) you can isolate a portion of your legacy code to migrate and call the equivalent .NET version of the code from your existing application—something like switching out an incandescent light bulb for one of those fluorescent ones. It should work if the lampshades still fits (and your spouse doesn't complain). The problem is that the new bulb emits RF noise that's picked up by your clock radio in the morning—one of those unexpected side-effects.

No, I'm not going to get into calling .NET code via COM interop¹ from VB6, but I *am* going to show you how to do something that has not been possible to do until now: create an ADO classic (ADOC) ADOc.Recordset from a DataTable.

What *can* be done?

Sure, you know that ADO.NET can read an existing ADOc Recordset object and generate a DataTable. Although you have to use the OleDb .NET Data Provider, it's really pretty easy. But this is a one-way street. There's no built-in mechanism to convert back to a Recordset.

```
Dim da As New OleDbDataAdapter
Dim dtFromRecordset As New DataTable
da.Fill(dtFromRecordset, rs)
MsgBox("Created DataTable " & _
& " with " & dtFromRecordset.Rows.Count & " rows.")
```

This means that you can have a .NET CLR application method (even using Visual Basic .NET) that can accept an ADOc Recordset as an input parameter. Performance notwithstanding, this solves one of the problems you might be worrying about. Yes, you could have manipulated the Recordset using COM interop in your .NET CLR app, but each object, property, or method reference has to traverse COM interop—and that's *really* going to impact performance. It might (just might) be faster to convert the data into a DataTable and back to a Recordset later. However, it might be smarter to get your ADOc code to call a .NET class (exposed as a COM method), pass in a query argument, and return a Recordset. I'm not going to debate the wisdom of this technique, but it might be your only viable recourse when attempting a partial migration to .NET.

What *couldn't* be done?

As I said before, what the OleDb provider *can't* do is convert a DataTable back to an ADOc Recordset. Fortunately, someone at Microsoft posted a "How To" on the MSDN site (KB article 31677) that does most of the work for us. Unfortunately, it's missing a feature or two that prevents the generated Recordset from containing some of the DDL information it could use. More importantly, though, the generated Recordset can't be updated or changed in any way. That's where I come in. Nope, I'm no XML expert (nor fan), but I was able to figure out how to tweak the original MSDN code and get it to work much better.

How it's done

The MSDN conversion routine (which I left basically intact) uses an XSL file to drive part of the transformation process. This is provided in the KB article, so I simply created a copy of the XML and saved it via Notepad to the solution directory. The rest of the Recordset XML is generated with a series of WriteAttributeString calls (a few are shown in Listing 1). This is where I found the first problem—the inability to write to the Recordset once it's reconstituted in VB6. After having compared the XML created with the ADOc Recordset Save method and the generated XML, I discovered many differences. One of the more subtle differences was the "writeunknown" attribute—it was missing from the NET-generated XML. According to Don Box² (and he should know), this attribute (*write* or *writeunknown*) is required when you want specify that the column is writeable. I added this

¹ See "Interoperating with Unmanaged Code" in Visual Studio .NET help.

² See "Using ADO to Create XML-based Recordsets" <http://msdn.microsoft.com/msdnmag/issues/0700/com/>

attribute and made a few other changes to make the .NET-generated XML more like the XML generated by ADOc. Without the “writeunknown” attribute, when you attempt to write to the Recordset (change a Field Value property) you get a rather bizarre error. “Multiple-step operation generated errors. Check each status value”. Sadly, there isn’t much documentation about this error. Annoyingly, several of the generated attributes were populated with the DataColumn properties that were less than useful. For example, the MaxLength property returned -1 in most cases. Fortunately, ADOc could still read the XML with far fewer attributes than generated by the conversion routine.

```
writer.WriteAttributeString("name", "", dc.ToString)
writer.WriteAttributeString("rs", "number", _
    "urn:schemas-microsoft-com:rowset", i.ToString)
writer.WriteAttributeString("rs", "writeunknown", _
    "urn:schemas-microsoft-com:rowset", "true")
    ' Added to permit RS updatability
writer.WriteAttributeString("rs", "baseCatalog", _
    "urn:schemas-microsoft-com:rowset", dbname)
writer.WriteAttributeString("rs", "baseTable", _
    "urn:schemas-microsoft-com:rowset", _
    dc.Table.TableName.ToString)
writer.WriteAttributeString("rs", "keycolumn", _
    "urn:schemas-microsoft-com:rowset", dc.Unique.ToString)
writer.WriteAttributeString("rs", "autoincrement", _
    "urn:schemas-microsoft-com:rowset", _
    dc.AutoIncrement.ToString)
```

Listing 1 Sample of WriteAttributeString code in the conversion routine.

The converter also includes a routine (HackADOXML) which walks the generated XML stream and makes some other changes. Apparently, (according to the documentation in the code) the “XSLT does not transform with full elements”. I expect they mean that the XSL transformation extrudes:

```
<root attr=""/>
```

instead of fully enumerated elements which are required by ADOc:

```
<root attr=""></root/>
```

Once the converter generates the XML, it’s saved to a temporary file and re-hydrated by the ADODB COM Recordset.Open method in the test harness This reconstituted Recordset can then be passed wherever you choose. Better yet, you can pass the XML version of the Recordset across firewalls where your legacy ADOc code can rehydrate it.

The test application

I wrote a test harness to convert a small DataTable to an XML Recordset and another Visual Basic 6.0 application to verify that the Recordset could be manipulated. I included code to modify the DataTable to ensure that the conversion routine left the changes intact—I wanted the uncommitted changes included in the Recordset—they were. (This VB.NET app is included in the download.) Listing 2 shows the routine to setup the test.

```
Private Sub btnCreateRecordset_Click(ByVal sender _
As System.Object, ByVal e As System.EventArgs) _
Handles btnCreateRecordset.Click
Try
    cn = New OleDbConnection("Provider=sqloledb;" _
        & "Data Source=ds;integrated ...
    da = New OleDbDataAdapter("SELECT AU_ID, Au_Lname, " _
        & " Au_Fname," & " State FROM Authors" _
        & " WHERE State = 'CA' ", cn)
    da.Fill(ds, "Authors")
    strResultXML = Application.StartupPath & _ _
        "\NewBaseRecordset.XML"
    rs = GenerateRecordset(ds, strResultXML)
    ' Convert base DataTable to XML Recordset. Make
    ' several changes to the DataSet to illustrate that
    ' the extruded XML includes these changes too.
    dt = ds.Tables("Authors") ' Address the table
    dr = dt.NewRow ' Create a new row
    dt.Rows.Add(New Object() {-1, "Flintstone", _
        "Fred", "UK"})
    dt.Rows.Add(New Object() {-2, "Flintstone", _
        "Wilma", "UK"})
    dt.Rows(2).Item("State") = "XX" ' Change something
```

```
strResultXML = Application.StartupPath + _
    "\UpdatedRecordset.XML"
rs = GenerateRecordset(ds, strResultXML) ' Convert
MsgBox("Converted DataTable to XML File")
btnCreateDataTable.Enabled = True
Catch ex As Exception
    MsgBox(ex.ToString)
Finally
End Try
End Sub
```

Listing 2. Generate a DataTable and call the conversion routine before and after changes.

The test harness routine used to call the KB-provided (and modified) conversion routine is available in the download . It constructs the path the XSL transform file and passes this path - as well as the path to the temporary XML result file along with the Database name - to the conversion routine. It does the conversion and returns a Recordset as well - populating the output XML file.

The modified conversion routine

The MSDN routine used to convert a DataTable to an ADOc Recordset is too large to print here, but I've included the source code in the downloadable zip file. The changes I made are clearly marked.

Summary

As time passes, we're finding more and more solutions to problems, issues and bugs uncovered as we attempt to migrate to .NET. Given the copious rewards of conversion, it's no wonder so many developers are making the attempt. Radical changes like this are (in my opinion) necessary from time to time to help purge the systems and applications of over-patched code that's not really doing what it's intended. It also gives developers a chance to learn and use new tools and techniques and discover a whole new set of issues, problems, bugs and things to fix. Isn't that what we get paid for?

download Bill1203.ZIP

bio on file