Connecting to SAP BW from ASP Using ADO MD

By Amyn Rajan, Dermot MacCarthy, Bruce Johnston

Simba Technologies Incorporated™

In a previous article, Connecting to SAP BW Using Visual Basic and ADO MD, we explained how to write custom Visual Basic applications that connect to SAP BW using the SAP BW OLE DB for OLAP (ODBO) Provider. We also covered ADO MD, Microsoft's extension to ADO for multi-dimensional data sources. The sample code that accompanied the article demonstrated the use of ADO MD and the SAP BW ODBO Provider with a "rich client" application developed with Visual Basic 6.0. In this article, we will take the same concepts and sample code and move them to a new environment – Active Server Pages (ASP).

If you have developed or maintained a web application on Windows during the past decade, chances are you are already familiar with ASP. Despite the introduction of ASP.NET a few years ago, ASP is unlikely to disappear any time soon. With that in mind, let's dig into the details of connecting to SAP BW from ASP via ODBO and MDX.

We assume you are familiar with the basics of HTML, ASP, Visual Basic Script (VBScript), and Microsoft Internet Information Services (IIS). You should also be familiar with ADO (ActiveX Data Objects), SAP BW, OLE DB for OLAP (ODBO), MDX, and OLAP concepts in general. We will be referring frequently to the article called "Connecting to SAP BW Using Visual Basic and ADO MD," so we will simply refer to it as the VB6 article. Reading it will help you to better understand the concepts presented here.

What is ADO MD?

ADO MD is an extension to ADO for retrieving multi-dimensional metadata and data from MDX-enabled OLAP data sources. It acts as a bridge between an application and an ODBO provider, and provides object models for browsing cube metadata and manipulating the results of MDX queries.

In this article, we will cover the particulars of some of the more common ADO MD objects, but by no means will we cover everything. The documentation for ADO MD is the best source for more detailed information. It is available on the Microsoft Developer Network here: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ado270/htm/ammscadomdfundamentals.asp.

ADO MD requires that the underlying data source have an ODBO provider. More details about the SAP BW ODBO provider can be found here: https://www.sdn.sap.com/irj/servlet/prt/portal/prtroot/com.sap.km.cm.docs/library/biw/2.0B%20OLE%20DB%20for%20OLAP.pdf.

What is OLE DB for OLAP (ODBO)?

OLE DB for OLAP is an extension of Microsoft's OLE DB standard that is designed for connecting to multi-dimensional data sources. It defines an API and a model for interaction between a "consumer" (application) and a "provider" (driver).

More information on OLE DB for OLAP is available on the Microsoft Developer Network here: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/oledbpart3_ole_db_for_olap.asp.

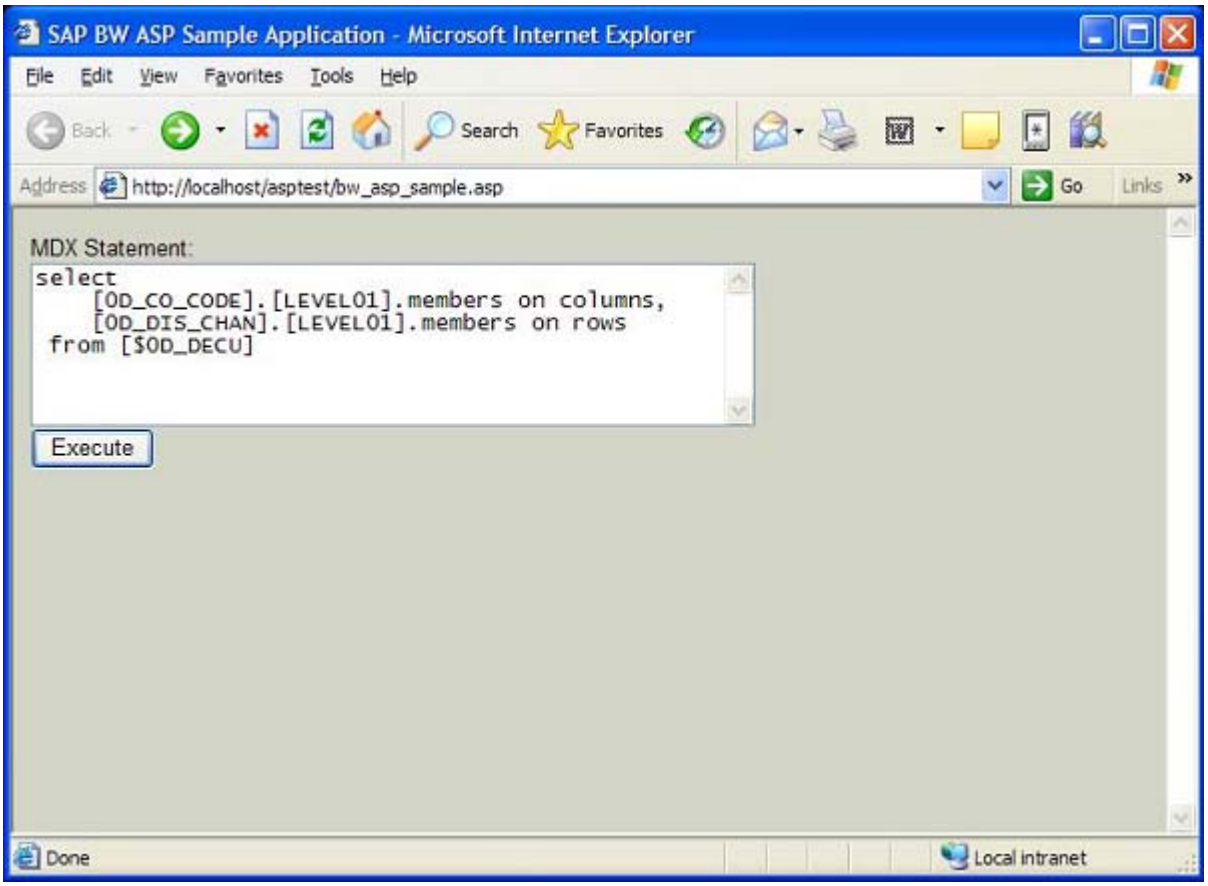
What is Multi-Dimensional eXpressions (MDX)?

MDX is a query language for multi-dimensional data sources. It is defined as part of the OLE DB for OLAP specification, and has also been adopted as the OLAP query language for XML for Analysis providers. MDX is to OLAP as SQL is to relational databases.

More information on MDX is available on the Microsoft Developer Network here: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/olapchapter_4_multidimensional_expressions.asp.

The Sample Web Application

The sample web application looks like this in Microsoft Internet Explorer:



The top of the page has a text box in which you can type an MDX statement. When you click the "Execute" button, the server-side script submits the query to the SAP BW ODBO provider. The provider returns the results, and the sample application renders them in a HTML table. The functionality of this sample application is basic, but the code illustrates all the principles of connecting via silent logon, executing an MDX query and retrieving the data.

Configuring the Sample Web Application

Before running the sample web application, you need to configure the web server. Since the sample code uses the SAP BW ODBO Provider, the first step is to install the provider on the web server by installing the SAP Frontend with the BW Add-on. Next, create a virtual directory using the IIS console that points to the directory containing the sample source files (the sample source should contain a file called `global.asa`, one `.asp` file, and one `.css` file). After this is done, you need to edit the `global.asa` file, shown here:

```
001: <script language="vbscript" runat="server">
002:
003: Option Explicit
004:
005: 'This is from oledb.h. It is one of the values of the
006: 'DBPROP_INIT_PROMPT property. DBPROMPT_NOPROMPT instructs
007: 'the provider to never display a logon UI.
008: Const DBPROMPT_NOPROMPT = 4
009:
010: Sub Application_OnStart
011:     'Connect to the SAP BW ODBO provider (Provider=MDrmSap).
012:     'Setting "Prompt" to DBPROMPT_NOPROMPT instructs the
013:     'provider to perform a silent logon.
014:     Application("sap_bw_connection_string") = _
015:         "Provider=MDrmSap;" & _
016:         "Data Source=<my_bw_server>;" & _
017:         "User ID=<my_user_id>;" & _
018:         "Password=<my_password>;" & _
019:         "SFC_CLIENT=100;" & _
020:         "SFC_LANGUAGE=EN;" & _
021:         "Prompt=" & _
022:         DBPROMPT_NOPROMPT
023: End Sub
024:
025: </script>
```

The `Application_OnStart` subroutine is only run the first time a client browser requests a page from the application's virtual directory. It sets up a connection string that will be used by the sample application when it needs to connect to SAP BW. The connection string consists of <property> = <value> pairs, separated by semicolons. There must be a `Provider` property in the connection string, and its value must be "MDrmSap" to connect to the SAP BW ODBO provider. The value of the `Prompt` property in the connection string is used to tell the provider to perform a silent logon. This is critical for a web application like this one – if it attempts to display a modal logon dialog box on the web server, the application will

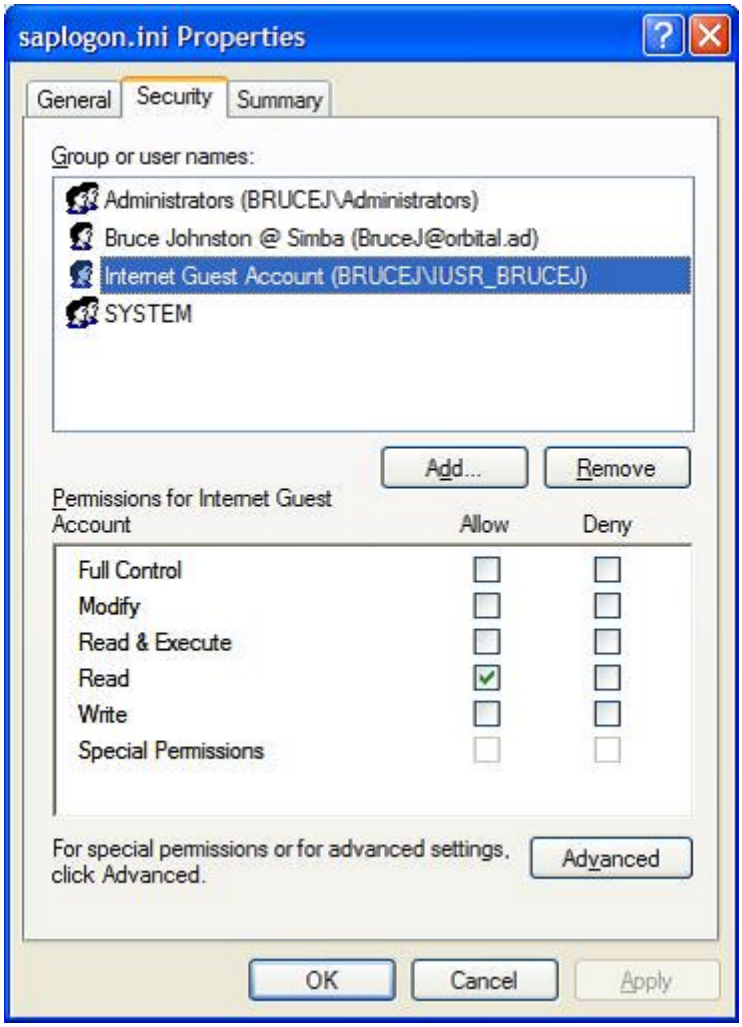
hang. The other possible values for the `Prompt` property are not useful for web applications, but if you're interested there is more information on this property in the VB6 article.

To enable the sample application to connect to your BW server, you need to replace the information on lines 16, 17, and 18 with the name of your BW system (the Description field from the SAP Logon utility) and your user ID and password. You may also need to adjust the `SFC_CLIENT` and `SFC_LANGUAGE` properties accordingly for your environment.

Once you have installed the SAP Frontend, configured IIS, and correctly set up the connection string in `global.asa`, the sample web application will run. However, it will not be able to connect to SAP BW until one more detail is taken care of. Since the sample code runs on a web server, it is subject to greater security restrictions than a rich-client application would be, especially if IIS is configured to use anonymous authentication. In this case, the sample code will run in the security context of the Internet Guest Account, `IUSR_`*`<MACHINE>`*, where *`<MACHINE>`* is the name of the web server. This account has limited access to the file system, which is an important detail, as we will see.

All clients that use the SAP OLAP BAPIs, including the SAP BW ODBO Provider, use the SAP Remote Function Call (RFC) library. Before it can connect to SAP BW, the RFC library needs to read the `SapLogon.ini` file. This file is created in the Windows directory (`c:\windows` or `c:\winnt`) when you install the SAP Frontend. By default, the `IUSR_`*`<MACHINE>`* account does not have access to files in the Windows directory for security reasons. So, in order for the sample web application to work, you must grant read permissions to this account for the `SapLogon.ini` file.

You can grant file permissions using Windows Explorer. Open the Windows folder (or Winnt folder), find `SapLogon.ini`, right-click on it, and click "Properties." Choose the "Security" tab and click the "Add..." button. Choose the "Internet Guest Account" or `IUSR_`*`<MACHINE>`* (this works in different ways on different versions of Windows) and click OK to add it to the list of user and group names. Next, select the account you just added in the list. Below the list of users and groups, you should see a list of permissions with "Allow" and "Deny" checkboxes next to them. Leave the "Deny" permissions as they are and make sure that only the "Read" permission is allowed. When you are done, the Properties dialog box should be configured like this:



Click OK.

Once this is done, you are ready to run the sample web application. Keep in mind that this same step will be necessary for each of your own web applications that connect to SAP BW.

Sample Web Application Structure

The VBScript code for the sample web application is contained in a single `.asp` file. There are a few variable and constant declarations at the top of the file, followed by several subroutines that contain most of the logic. The HTML content and the scripts embedded within it are at the bottom of the file.

In order to understand the overall structure and control flow of the sample web application, it's best to start at the bottom with the HTML portion. Note that the line numbers in the code below are for easy reference in this article. They do not correspond to line numbers in the actual file.

```
001: <html>
002: <head>
003: <title>SAP BW ASP Sample Application</title>
004: <link href=”simbastyle.css” rel=”Stylesheet”/>
005: <style type=”text/css”></style>
006: </head>
007: <body>
008:
009: <%
010: Dim mdxQuery
011: mdxQuery = Request.Form(“mdxQuery”)
012:
013: Dim isExecuteClicked
014:
015: If mdxQuery <> “” Then
016:     isExecuteClicked = True
017: Else
018:     isExecuteClicked = False
019:     mdxQuery = _
020:         “select” & _
021:         vbNewLine & _
022:         “    [0D_CO_CODE].[LEVEL01].members on columns,” & _
023:         vbNewLine & _
024:         “    [0D_DIS_CHAN].[LEVEL01].members on rows” & _
025:         vbNewLine & _
026:         “ from [$0D_DECU]”
027: End If
028: %>
029:
030: <form action=”bw_asp_sample.asp” method=”post”>
031:     MDX Statement:
032:     <br/>
033:     <textarea name=”mdxQuery” cols=”50" rows=”7">
034: <% Response.Write Server.HTMLEncode(mdxQuery) %>
035:     </textarea>
036:     <br/>
037:     <input type=”submit” value=”Execute”>
038: </form>
039:
040: <%
041: If isExecuteClicked Then
042:     ExecuteButton_Click
043:
044:     Dim row, col
045: %>
046: <table border=”1">
047: <% For row = 0 To FixedRows - 1 %>
048:     <tr>
049:     <% For col = 0 To Cols - 1 %>
050:         <th><% Response.Write ResultsGrid(row, col) %></th>
051:     <% Next %>
052:     </tr>
053: <% Next %>
054: <% For row = FixedRows To Rows - 1 %>
055:     <tr>
```

```
056:     <% For col = 0 To FixedCols - 1 %>
057:         <th><% Response.Write ResultsGrid(row, col) %></th>
058:     <% Next %>
059:     <% For col = FixedCols To Cols - 1 %>
060:         <td><% Response.Write ResultsGrid(row, col) %></td>
061:     <% Next %>
062:     </tr>
063: <% Next %>
064: </table>
065: <%
066: End If
067: %>
068:
069: </body>
070: </html>
```

First, a quick word on style: There is a `<link>` tag on line 4 that refers to a `.css` file containing the style sheet used by the sample web application. This `.css` file is included with the sample code, but is not presented here since its only purpose is to make the sample web application easier on the eyes.

When using the sample web application from a browser, this ASP script is run on the server the first time you navigate to the page, and again every time you click the "Execute" button. The script beginning on line 10 detects the difference between these two cases by looking for an item named "mdxQuery" in the POST data sent with the HTTP request. The first time you navigate to the page, this item does not exist and `isExecuteClicked` is set to `False`. After you click "Execute," the next HTTP request sent to the server contains the contents of the "mdxQuery" `<textarea>` in the POST data, so `isExecuteClicked` will be set to `True` in this case.

The "mdxQuery" `<textarea>` is part of the `<form>` beginning on line 30. Its contents again depend on whether or not "Execute" was just clicked. If not, the default MDX statement from lines 19 through 26 is used. Otherwise, the MDX statement from the POST data is used. In a sense, the ASP script "invokes" itself when "Execute" is clicked, passing the contents of its own `<textarea>` from the browser into the contents of the new invocation's `<textarea>`, and back to the browser again. This gives the user interface the illusion of persistence, even though in reality the state of the UI is being passed back and forth between the browser and the web server.

Line 41 is where things get more interesting. In the sample application from the VB6 article, the results of an MDX query were displayed in a `MSFlexGrid` control. Since this is a web application, we no longer have that option. However, in the interest of re-using code, the sample web application "emulates" a `MSFlexGrid` using a bit of script and a two-dimensional array called `ResultsGrid`. The `Cols` and `Rows` variables describe the size of `ResultsGrid`. The `FixedCols` and `FixedRows` variables describe the number of row and column headings in the grid. All of these variables are initialized as a result of the call

to `ExecuteButton_Click` on line 42, and are subsequently used in lines 46 through 64 to render the grid as an HTML table.

The variables used to hold the results are declared at the top of the file. Their declarations look like this:

```
001: <%
002: Option Explicit
003:
004: ‘This is defined as part of ADO, but for VBScript we must
005: ‘define it ourselves.
006: Const adStateOpen = 1
007:
008:
009: ‘These define the size of the table that will be used to
010: ‘display the Cellset. They are analogous to their
011: ‘corresponding properties in the MSFlexGrid control.
012: Dim Cols       ‘Total number of columns in the grid.
013: Dim Rows       ‘Total number of rows in the grid.
014: Dim FixedCols  ‘Number of columns containing headings.
015: Dim FixedRows  ‘Number of rows containing headings.
016:
017:
018: ‘This is a two-dimensional array that is used to emulate
019: ‘the structure of an MSFlexGrid. It will be sized later
020: ‘according to the shape of the Cellset to be displayed.
022: Dim ResultsGrid()
```

Now that you have the big picture of what the sample web application is doing, it’s time to look at the details.

Connecting to the SAP BW ODBO Provider

`ExecuteButton_Click` invokes most of the code in the sample application either directly or indirectly. Here is the code:

```
001: ‘This subroutine is invoked when the user clicks “Execute”.
002: ‘It connects to the SAP ODBO provider, then executes an MDX
003: ‘statement and displays the results.
004: Sub ExecuteButton_Click()
005:     On Error GoTo 0
006:     Dim conn
007:     Set conn = Server.CreateObject(“ADODB.Connection”)
008:
009:     ‘Use the connection string that was set in global.asa.
010:     conn.Open Application(“sap_bw_connection_string”)
011:
012:     ExecuteAndFormatResults conn
013:     conn.Close
014: End Sub
```

This subroutine contains all the logic for connecting to SAP BW. (Unlike the sample application from the VB6 article, error handling here is virtually non-existent. This is a limitation of VBScript – proper error handling in ASP is beyond the scope of this article.) The bulk of the work is done by the `ExecuteAndFormatResults` subroutine, which is called on line 12. We will cover this subroutine in detail later, but for now, let's look more closely at the process of connecting to SAP BW.

Since ADO MD is a superset of ADO, you use the familiar ADO objects for connecting to the ODBO provider. `ExecuteButton_Click` creates an `ADODB.Connection` object and assigns it to the local variable `conn` on line 7. Then on line 10, it opens the connection by passing a connection string to the `Open` method. The connection string comes from the ASP `Application` object. Recall that it was stored there by the `Application_OnStart` subroutine in the `global.asa` file.

Next, we will look in detail at the `ExecuteAndFormatResults` subroutine, which executes an MDX statement, retrieves the result as a `Cellset`, and copies the results into the `ResultsGrid` array.

Executing a MDX Statement and Opening a Cellset

The sample application uses the `ADOMD.Cellset` object to execute an MDX statement and hold the multi-dimensional results. `Cellset` objects have an `Open` method, which executes the MDX statement given by the `Cellset`'s `Source` property against the connection given by the `Cellset`'s `ActiveConnection` property. For convenience, `Open` takes optional parameters that allow `Source` and `ActiveConnection` to be set at the time `Open` is called. This is shown on line 14 of the `ExecuteAndFormatResults` subroutine shown below:

```
001: 'This subroutine executes the MDX statement given in the
002: 'form, obtains the multidimensional results in a Cellset,
003: 'then renders the contents of the Cellset as HTML.
004: Sub ExecuteAndFormatResults(conn)
005:     Dim csResults
006:     Set csResults = Server.CreateObject("ADOMD.Cellset")
007:
008:     Dim mdxQuery
009:     mdxQuery = Request.Form("mdxQuery")
010:
011:     'Detect syntax errors and the like.
012:     On Error Resume Next
013:
014:     csResults.Open mdxQuery, conn
015:
016:     'Must guard against closed Cellsets. ADOMD will return a
017:     'closed Cellset when the provider doesn't create a result
018:     '(e.g. – when an MDX CREATE or DROP statement is executed).
019:     'Also, due to the limited error handling in VBScript, this
020:     'is the only way we can detect an error when executing the
021:     'MDX statement.
```

```
022:     If csResults.State <> adStateOpen Then
023:         Response.Write “<p>The MDX statement did not “ & _
024:                        “produce a result, or was incorrect.</p>”
025:         Exit Sub
026:     End If
027:
028:     ‘Restore the “panic” error handling mode.
029:     On Error Goto 0
030:
031:     ‘This is just a sample application, so we won’t bother
032:     ‘dealing with more than two axes. In order to render three or
033:     ‘more axes to a two-dimensional grid, you would have to
034:     ‘implement your own “flattening” algorithm to combine axes
035:     ‘together for display purposes.
036:     If csResults.Axes.Count <= 2 Then
037:         ResizeGridToCellset csResults
038:         PopulateGridFromCellset csResults
039:     Else
040:         Response.Write “<p>Cellsets with more than two axes “ & _
041:                        “are not supported.</p>”
042:     End If
043:     csResults.Close
044: End Sub
```

The first parameter of the `Cellset`'s `Open` method is `Source`, which in our case is the MDX query text from the form. The second parameter is `ActiveConnection`, which is the parameter `conn` passed to the subroutine.

After calling `Open`, `ExecuteAndFormatResults` performs an additional test at line 22 to make sure that the `Cellset` is actually open. This is necessary because of the limited error handling capability of VBScript. It would be bad to generate a run-time error if the user types an invalid MDX statement – the result would be the dreaded HTTP 500 “internal server error” page. Instead, the error-handling mode is changed on line 12 to resume at the next statement in case of an error. This means that the only way to know that an error occurred is to see if the `Cellset` is open or not. Finding a more elegant solution is left as an exercise for the reader.

The test on line 22 is also necessary because certain MDX statements, such as `CREATE MEMBER`, `CREATE SET`, etc., do not return results. The result in terms of ADO MD is a `Cellset` object that is still closed, even though the `Open` method succeeded on it. If this is the case, `ExecuteAndFormatResults` writes an error message to the HTML output at lines 23 and 24, and then exits at line 25. Otherwise, it prepares the contents of the `Cellset` for display.

Once the `Cellset` has been opened, `ExecuteAndFormatResults` checks to see how many axes it has (line 36). Each `Axis` object in the `Cellset`'s `Axes` collection corresponds to one of the axis specifications in the original MDX statement. For example, consider the following MDX statement:

```
select
    [0D_CO_CODE].[LEVEL01].members on columns,
    [0D_DIS_CHAN].[LEVEL01].members on rows
 from [$0D_DECU]
```

The `Cellset` corresponding to this MDX statement contains two axes in its `Axes` collection – `Axes(0)`, which contains all the company codes from the `0D_CO_CODE` characteristic, and `Axes(1)`, which contains all the distribution channels from the `0D_DIS_CHAN` characteristic.

If a `Cellset` has at most two axes, it is easy to map the `COLUMNS` and `ROWS` axes to the row and column headings of the HTML table. If there were more than two axes, then we would have to combine axes together on the rows or columns of the grid and index into the cell data accordingly. To keep things simple, our sample web application can only display the contents of `Cellsets` with at most two axes.

`ResizeGridToCellset` and `PopulateGridFromCellset` resize the `ResultsGrid` array and copy the `Cellset` content into it, respectively. These subroutines are covered in the next few sections.

Resizing ResultsGrid to Contain a Cellset

In the VB6 article, we used a `MSFlexGrid` control to display the contents of the `Cellset`. One of the useful features of this control is its ability to automatically merge duplicate cells together. In the sample web application, we have to render the grid as HTML "by hand," so to keep things simple, we won't bother trying to merge duplicate cells.

For example, consider the following MDX query:

```
select
    {[Measures].DefaultMember} on columns,
    CrossJoin( [0D_CO_CODE].[LEVEL01].members,
               [0D_DIS_CHAN].[LEVEL01].members ) on rows
 from [$0D_DECU]
```

For the sake of comparison with the VB6 article, here is what the sample web application looks like after executing this query:

The `ResizeGridToCellset` subroutine is responsible for determining the layout of the `ResultsGrid` array and the corresponding HTML table like the one shown above. It determines the layout based on the axes of the `Cellset` object. This code is adapted from the VB6 article, so if you've read that article, this should all look familiar:

```
001: ‘Re-sizes the ResultsGrid array to accommodate the results in the
002: ‘given Cellset. Assumption: The Cellset contains at most two axes.
003: Sub ResizeGridToCellset(csResults)
004:     ‘These variables will keep track of the size of the
005:     ‘COLUMNS and ROWS axes. The size of an axis set is
006:     ‘really measured in two ways: by the number of tuples it
007:     ‘contains, and by the number of dimensions nested on the
008:     ‘axis.
009:     Dim rowsAxisDimensionCount, rowsAxisTupleCount
010:     Dim colsAxisDimensionCount, colsAxisTupleCount
011:
012:     rowsAxisDimensionCount = 0
013:     rowsAxisTupleCount = 0
014:     colsAxisDimensionCount = 0
015:     colsAxisTupleCount = 0
016:
017:     ‘These next two values are always either 0 or 1, to
018:     ‘ensure that there is room to display data even when one
019:     ‘of the axes is missing or empty.
020:     Dim extraRow, extraColumn
021:
022:     extraRow = 0
023:     extraColumn = 0
024:
025:     ‘If there is a COLUMNS axis...
026:     If csResults.Axes.Count > 0 Then
027:         ‘Note that “Positions” are the same as “Tuples”.
028:         ‘This is just a terminology discrepancy between
029:         ‘ADOMD and MDX.
030:         colsAxisTupleCount = csResults.Axes(0).Positions.Count
031:         colsAxisDimensionCount = csResults.Axes(0).DimensionCount
032:
033:         ‘If there is a ROWS axis...
034:         If csResults.Axes.Count > 1 Then
035:             rowsAxisTupleCount = csResults.Axes(1).Positions.Count
036:             rowsAxisDimensionCount = _
037:                 csResults.Axes(1).DimensionCount
038:         End If
039:     End If
040:
041:     ‘If the rows axis is missing or empty, we still want to
042:     ‘ensure that there is a single non-fixed row to display
043:     ‘data (if ROWS is missing) or status information (if
044:     ‘ROWS is empty).
045:     If rowsAxisTupleCount = 0 Then
046:         extraRow = 1
047:     End If
048:
049:     ‘If the columns axis is missing or empty, we still want
050:     ‘to ensure that there is a single non-fixed column to
051:     ‘display data (if COLUMNS is missing) or status
052:     ‘information (if COLUMNS is empty).
053:     If colsAxisTupleCount = 0 Then
054:         extraColumn = 1
055:     End If
```

```
056:
057:     'The following arithmetic is carefully crafted to work in
058:     'all cases. For details, see the companion article.
059:     Rows = colsAxisDimensionCount + rowsAxisTupleCount + extraRow
060:     Cols = rowsAxisDimensionCount + colsAxisTupleCount + _
061:            extraColumn
062:     FixedRows = colsAxisDimensionCount
063:     FixedCols = rowsAxisDimensionCount
064:
065:     'Re-size the array into which the cell data will
066:     'be copied.
067:     ReDim ResultsGrid( Rows, Cols )
068: End Sub
```

The `ResizeGridToCellset` subroutine is self-explanatory up to about line 25. At that point, the code determines the number and size of the axes in the `Cellset`. The number of tuples in an `Axis` object can be obtained from the `Count` property of its `Positions` collection. (Note that "position" in ADO MD is really a synonym for "tuple".) The number of dimensions in an `Axis` is given by its `DimensionCount` property. The code between lines 25 and 39 obtains the dimension and tuple counts for the `ROWS` and `COLUMNS` axes of the `csResults Cellset`. If either axis is missing or empty, its tuple and dimension counts will remain zero. Note that it is not possible for `COLUMNS` to be missing unless `ROWS` is also missing (i.e. – all axes must be contiguous).

The next several lines, from 41 to 55, deal with special cases where one or both of the axes are missing or empty. This all culminates in the arithmetic starting on line 57, which ensures that:

- There are enough rows in the grid to contain headings for the `COLUMNS` axis and the entire contents of the `ROWS` axis. If the `ROWS` axis is missing, then there must be one non-fixed row to contain the cell data.
- There are enough columns in the grid to contain headings for the `ROWS` axis and the entire contents of the `COLUMNS` axis. If the `COLUMNS` axis is missing, then there must be one non-fixed column to contain the cell data.
- There are enough fixed rows in the grid to contain headings for the `COLUMNS` axis.
- There are enough fixed columns in the grid to contain headings for the `ROWS` axis.

"Fixed" columns and rows in this case are stored as regular columns and rows in the `ResultsGrid` array, but they will be rendered as `<th>` elements instead of `<td>` elements in the HTML table.

The following cases are handled by this code:

- Case 1: There are non-empty `ROWS` and `COLUMNS` axes.
- Case 2: There is a non-empty `COLUMNS` axis, but no `ROWS` axis.
- Case 3: There are no axes.
- Case 4: There are both `ROWS` and `COLUMNS` axes, but the `ROWS` axis is empty.

- Case 5: There are both ROWS and COLUMNS axes, but the COLUMNS axis is empty.
- Case 6: There is only a COLUMNS axis, and it is empty.

Verifying that the sample web application works with all of the above cases is left as an exercise for the reader.

The final step on line 67 is to re-size the ResultsGrid array so that it can hold the contents of the Cellset.

As you can see, dealing with multi-dimensional data can be complicated, even with at most two axes. The good news is that once you implement this kind of logic, it should be re-usable in other applications, as we have shown with the sample application from the VB6 article, and with this sample web application.

Populating ResultsGrid from the Cellset Contents

Once the ResultsGrid has been resized, we are ready to populate it with data from the Cellset. The PopulateGridFromCellset subroutine does this. This code is also adapted from the VB6 sample application and will look familiar, if you have read the VB6 article:

```
001: ‘Populates the ResultsGrid from the contents of the given Cellset.
002: ‘Assumption: The size of the ResultsGrid has already been
003: ‘calculated appropriately. This is a requirement because a lot is
004: ‘inferred about the structure of the Cellset based on the
005: ‘structure of the ResultsGrid.
006: Sub PopulateGridFromCellset(csResults)
007:     ‘Populate the fixed rows and columns first.
008:     Dim col, row
009:
010:     ‘Column headings are in the fixed rows.
011:     For row = 0 To FixedRows - 1
012:         ‘Fill in headings above non-fixed columns.
013:         For col = FixedCols To Cols - 1
014:             ‘Assumptions: If there are fixed rows, then there must
015:             ‘be a COLUMNS axis. If there is a COLUMNS axis, then
016:             ‘the number of dimensions on it (and therefore the
017:             ‘number of members in each tuple) is equal to the
018:             ‘number of fixed rows. The number of tuples on the
019:             ‘COLUMNS axis is equal to the number of non-fixed
020:             ‘columns.
021:             Dim colTuple
022:             Set colTuple = _
023:                 csResults.Axes(0).Positions(col - FixedCols)
024:             ResultsGrid(row, col) = colTuple.Members(row).Caption
025:         Next
026:     Next
027:
028:     ‘Row headings are in the fixed columns.
029:     For col = 0 To FixedCols - 1
030:         ‘Fill in headings to the left of the non-fixed rows.
031:         For row = FixedRows To Rows - 1
```

```
032:             ‘Assumptions: If there are fixed columns, then there
033:             ‘must be a ROWS axis. If there is a ROWS axis, then
034:             ‘the number of dimensions on it (and therefore the
035:             ‘number of members in each tuple) is equal to the
036:             ‘number of fixed columns. The number of tuples on the
037:             ‘ROWS axis is equal to the number of non-fixed rows.
038:             Dim rowTuple
039:             Set rowTuple = _
040:                 csResults.Axes(1).Positions(row - FixedRows)
041:             ResultsGrid(row, col) = rowTuple.Members(col).Caption
042:         Next
043:     Next
044:
045:     ‘Prepare to handle cell retrieval errors if one of the
046:     ‘axes is empty.
047:     On Error Resume Next
048:
049:     ‘The default cell contents will be used if an error occurs
050:     ‘fetching a cell from the Cellset. This should only occur
051:     ‘when there are no cells to fetch.
052:     Dim cellStr
053:     cellStr = “No cell data”
054:
055:     ‘Fill in the cell data. It goes in the non-fixed columns
056:     ‘and rows.
057:     For col = FixedCols To Cols - 1
058:         For row = FixedRows To Rows - 1
059:             ‘The cardinality of the cell co-ordinate we pass
060:             ‘to the Cellset must be correct depending on how
061:             ‘many axes are present.
062:             If FixedCols <> 0 And FixedRows <> 0 Then
063:                 ‘Assumption: Both ROWS and COLUMNS axes are
064:                 ‘present and non-empty.
065:                 cellStr = _
066:                     csResults(col - FixedCols, _
067:                               row - FixedRows).FormattedValue
068:             ElseIf FixedRows <> 0 Then
069:                 ‘Assumption: There is only a COLUMNS axis
070:                 ‘present, and it is non-empty.
071:                 cellStr = csResults(col).FormattedValue
072:             Else
073:                 ‘Assumption: Both axes are missing or empty.
074:                 ‘Rather than try to detect the empty axes, we
075:                 ‘can just catch the error and put a status
076:                 ‘message in the grid.
077:                 cellStr = csResults(0).FormattedValue
078:             End If
079:
080:             ResultsGrid(row, col) = cellStr
081:         Next
082:     Next
083: End Sub
```

This subroutine has three sets of loops: Two populate the “fixed” rows and columns with headings, and the third populates the “center” of `ResultsGrid` with cell data.

The `For` loop that begins on line 10 iterates through the fixed rows at the top of the grid. For each fixed row, the inner `For` loop populates each non-fixed column with the member caption from the corresponding member of the current tuple of the `COLUMNS` axis. The comments in the code detail the assumptions that are made about the layout of the grid. It is important that these assumptions are valid, since they allow this code to handle all six cases outlined in the previous section. The `For` loop beginning on line 28 is just the converse of the loop on line 10. Swap the words “row” and “column” and you’ll see that it works in exactly the same manner, populating row headings with captions from the `ROWS` axis.

The set of nested loops that populate the grid with cell data begins on line 55. These loops iterate through all the intersections of non-fixed columns and rows. For each intersection, the formatted value of the corresponding cell is fetched from the `Cellset` object. The default property of the `Cellset` is an indexed property that contains the cell data. There are a few different types of indices that this property will accept, but the one we use is simply an *n*-dimensional co-ordinate. The trick is to determine what *n* is supposed to be. Once again, this is done based on assumptions about how the `ResultsGrid` was resized earlier. The `If-Else` spanning lines 59 through 78 implement this logic.

An important corner case to understand is where one or more of the axes of a `Cellset` are empty (i.e. – contain no tuples). In this case, the `Cellset` contains no cell data and any attempt to use its indexed property will result in a run-time error. The `PopulateGridFromCellset` subroutine deals with this by resuming at the statement after the error, which will fill the cell with a status message. This seems like a better solution than not displaying anything at all, although it is less elegant than the error handling implemented in the VB6 article’s sample application. Once again, our choices are limited by the more primitive error-handling facilities of VBScript.

Conclusion

In this article, we have presented a sample ASP application written in VBScript with ADO MD. It connects to SAP BW via the SAP BW OLE DB for OLAP (ODBO) provider, executes an MDX statement, and displays the results in an HTML table. The sample web application is simplistic, but it illustrates all the concepts you need to know to be able to write your own ASP applications to connect to SAP BW.

Download Sample Code

Download sample code for the full application here:
http://www.simba.com/Knowledge_Center/KnowledgeCenter.htm

About Simba Technologies

Simba Technologies Incorporated (http://www.simba.com) builds development tools that make it easier to connect disparate data analysis products to each other via standards such as, ODBC, JDBC, OLE DB, ODBO (OLE DB for OLAP), and XML for Analysis. Independent software vendors that want to extend their proprietary architectures to include advanced analysis capabilities look to Simba for strategic data connectivity solutions. Customers use Simba to leverage and extend their proprietary data through high performance, robust, fully customized, standards-based data access solutions that bring out the strengths of their optimized data stores. Through standards-based tools, Simba solves complex connectivity challenges, enabling customers to focus on their core businesses.

About the Authors

Amyn Rajan is President and CEO of Simba Technologies Incorporated. Amyn has over 14 years experience in custom software development, and is responsible for driving Simba's success as a leader in data connectivity solutions.

Dermot MacCarthy is a Senior Computer Scientist with Simba Technologies. Dermot has over 25 years experience in software and technology, specializing in database access and connectivity solutions.

Bruce Johnston is a Computer Scientist with Simba Technologies. Bruce has over six years experience in software development, specializing in OLAP and data access.

Prepared for SAP Developer Network

Simba Technologies Incorporated

1090 Homer Street, Suite 200

Vancouver, BC Canada

V6B 2W9

Tel. 604.633.0008 Ext. 2

Fax. 604.633.0004

Email. solutions at simba.com

www.simba.com