.NET Interop for Visual FoxPro Applications

By Rick Strahl
http://www.west-wind.com/

Last Update: September 23, 2002

Code for this article:
http://www.west-wind.com/presentations/VfpDotNetInterop/VfpDotNetInterop.zip

This document covers:

Note: The code listings in this article use C# for .NET code and Visual FoxPro for COM or client code.

Now that .NET is here you've undoubtedly have the urge to use or at least play with the new functionality that the platform provides. Unfortunately migrating to .NET from Visual FoxPro (or most other development languages) is a big step that requires a steep learning curve. Integration between the old and the new will be crucial as a first step to provide for the ramp up time that's needed to get up to speed on the new platform as well as providing vital links between old and new applications. In this article Rick looks at the most common ways that you can use to integrate logic and data between Visual FoxPro and .NET.

Like any new development platform or environment, .NET introduces a brand new environment and many new concepts for developing applications. To us as developers this usually means that there's a large learning curve involved in understanding the new platform, finding the the most efficient ways of performing common tasks and working around the limitations of the new environment. Although Microsoft will be pushing hard to convert as many developers to .NET as quickly as possibility, it's obvious that the transition to an all .NET world will take a fair amount of time to accomplish.

For this reason, it's vital that there are mechanisms readily available to share non-managed .NET code with managed .NET code and vice versa. If you've been reading the trade press, you've undoubtedly heard that Web Services are one of the biggest features of .NET and one of the big selling points of Web Services of course is the fact that they can be accessed by many different kinds of development environments both old and new. However, Web Services are only one of the options available for interoperating and probably the least likely to be useful for extending the life of **old** applications.

As in previous updates of development environments data access through standard data access components make it possible to directly use data from various different data sources including Visual FoxPro. Visual FoxPro 7 sports a new OleDb driver which improves significantly on some of the limitations of the VFP ODBC driver and makes it a capable provider for accessing data directly from .NET applications even if there are a number of features that are useful for .NET that are missing.

Prior to .NET, COM has been the primary interop mechanism that applications could use to communicate amongst each other and .NET provides extensive support for COM. You can both access COM components from .NET and expose .NET components through COM through its Runtime Callable Wrapper (RCW) which provides a COM wrapper around the .NET runtime.

I'll go over each of these approaches and in the process point out some issues that you need to watch out for. As all tools geared toward providing backwards compatibility they rarely provide all the functionality that was provided by the original implementations, so some things require special attention or changes to properly work in a .NET interop environment.

Although this article discusses Interop mechanisms in general I'll focus more specifically on ASP.NET as this is likely the most used platform of .NET technology at this point. Keep in mind that the same principles apply to Windows Forms or even Console applications. ASP.NET however introduces some additional considerations because it is a server based technology and requires special considerations for performance and scalability. So I'll use ASP.NET in this article as my vehicle to describe most of these technologies.

Accessing VFP Data with OleDb

This hardly qualifies as an eye opening topic, but OleDB support in Visual FoxPro 7.0 has been significantly updated over the existing ODBC implementation that makes it more realistic to use VFP data via a generic driver, especially in an ASP.NET Web environment.

By using the OleDb driver you have the ability to use ADO.NET and the DataSet style implementation given some limitations that I'll discuss later on. Performance of the driver is adequate although as we'll see still a bit of a bottleneck compared to native performance using pure Visual FoxPro code or code loaded from COM objects. The new OleDb provider is also multi-threaded and can run multiple queries simultaneous, which was probably the biggest problem with the ODBC driver when used in ASP and now ASP.NET applications.

To demonstrate the basic principles of OleDb access against a Fox data source I'll build a small and simplistic sample ASP.NET Web Form applet that lets you browse and edit entries in the TASTRADE sample database shipped with Visual FoxPro.

File Access requirements for ASP.NET

If you plan to use ASP.NET to access VFP via OleDb you have to make sure that your data directories are properly configured to allow the ASPNET user full access to your data paths. ASP.NET runs under this user by default although you can change the actual account in Web.config if you choose. Whatever account you use this account must be able to have full NTFS access rights in the data directories where data is read from and written to. You can use Explorer and the Security tab to accomplish this.

Simple DataSet retrieval

But before I do this let me just give you a small example that demonstrates the basics of creating a DataSet with VFP data on an ASP.NET form. This example consists of a very simple ASP.NET page that contains only two controls: a label to show error information (lblErrorMsg) and a DataGrid (oCustList) to display the data from a query against VFP data. The following code is the CodeBehind page that uses a separate GetCustomerList() method on the form to perform the data retrieval into a DataSet (Listing 1).

Listing 1 (C# ASP.NET): Retrieving a VFP OleDb result into an ASP.NET DataSet

```
…
using System.Data.OleDb;

namespace ASPInterOp
{
   public class SimpleDataSet : System.Web.UI.Page
   {
      OleDbConnection oConn = null;
      DataSet oDS = null;

      protected System.Web.UI.WebControls.DataGrid oCustList;
      protected System.Web.UI.WebControls.Label lblErrorMsg;

      private void Page_Load(object sender, System.EventArgs e)
      {

         if ( this.GetCustomerList("CustomerList") )
         {
            this.oCustList.DataSource = this.oDS.Tables["CustomerList"];
            this.oCustList.DataBind();
         }
      }

      private bool GetCustomerList(string lcCursor)
      {
         // Put user code to initialize the page here
         this.oConn = new OleDbConnection("Provider=vfpoledb.1;" +
             @"Data Source=D:\Programs\vfp70\Samples\Data\testdata.dbc;" +
              "Exclusive=false;Nulls=false");

         try {
            this.oConn.Open();
         }
         catch(Exception ex)
         {
            this.lblErrorMsg.Text = ex.Message;
            return false;
         }

         string lcSQL = "select company, contact, city from customer";
         OleDbDataAdapter oAdapter = new OleDbDataAdapter(lcSQL,this.oConn);

         if (this.oDS == null)
         {
            this.oDS = new DataSet();
         }

         try
         {
            oAdapter.Fill(this.oDS,lcCursor);
         }
         catch(Exception ex)
         {
            this.lblErrorMsg.Text = ex.Message;
            return false;
         }
         finally
         {
            this.oConn.Close();
         }

         return true;
      }
}
```

As you can see it takes a fair amount of code in the GetCustomerList() method to do this, but the code is pretty straight forward. You start with a connection object that is passed an OleDb connection string that points at the VFP OleDb driver and a path to a DBC or a free table directory (if you don't include a DBC tables are treated as free tables).

Next a DataAdapter is created. The DataAdapter is the 'data retrieval' object. It knows how to talk to the data source and return the data in a specified format. The adapter is specific to the backend used, just as the connection is. Hence the class name is **OleDb**DataAdapter. If you go against SQL Server you can use the **Sql**DataAdapter instead, which is the SQL Server specific implementation that bypasses OleDb. With VFP you have to use the OleDb adapter and connection objects.

The adapter acts as the intermediary that's used to query the data from the database and return a result to the application. Natively a DataAdapter returns a DataReader, which is basically a stream of forward only data. Using the DataReader to retrieve the data is the most efficient way to get the data if you only need to display data and don't need any special filtering or jump through the data.

The DataReader stream is also used internally by the DataAdapter and can be used to create other objects which can populate their own data control objects. One of these and probably the most important one in .NET is the DataSet. In this case I'm using the DataAdapter's Fill() method to 'fill' the data into a DataSet object. Notice that the DataSet is not prefixed by OleDb or Sql. This is because a DataSet is a generic object that is completely disconnected from the parent datasource. Think of the DataSet as a container for multiple cursors – DataSets can run and hold results from multiple queries simultaneously. So when a new query is executed you need to tell the DataAdapter what name to give the table created into the DataSet.

Once a DataSet has been created you can access its data directly either using programmatic access or by databinding to a data aware control. Many of the user interface controls in .NET have intrinsic support for DataSets. Here I'm using a DataGrid to bind the data and simply display a simple list of the data in an HTML table through the DataGrid Server Control object (oCustList) by using:

```
this.oCustList.DataSource = this.oDS.Tables["CustomerList"];
```

DataSets are made up of Tables, Rows and Columns (and a few other things like Relations and Constraints). Here I'm binding to a table in the DataSet. Tables inside of a DataSet have an automatic data view that represents a layout that determines how the table renders into the DataGrid. The default iew is very basic and displays all the fields in the table in a grid format with the field names as headers. The result is very raw, but useful for quickly seeing the data displayed (Figure 1). To customize the view you have to set properties on the DataGrid either in the <asp:DataGrid> tag or by setting properties on the object. I'm not going into the details of the datagrid here but rather focus on the data access issues.

You can also access the data directly via code. A DataSet contains a collection of Tables, which in turn contains a collection of Rows. The Rows then contain the individual fields. The following snippet demonstrates:

```
DataRow loRow = this.oDS.Tables["CustomerList"].Rows[0];  // first data row
int lnRowCount = this.oDS.Tables["CustomerList"].RowCount;

string lcCompany = loRow["company"].ToString();
decimal MaxOrdAmt = (decimal) loRow["maxordamt"];
```

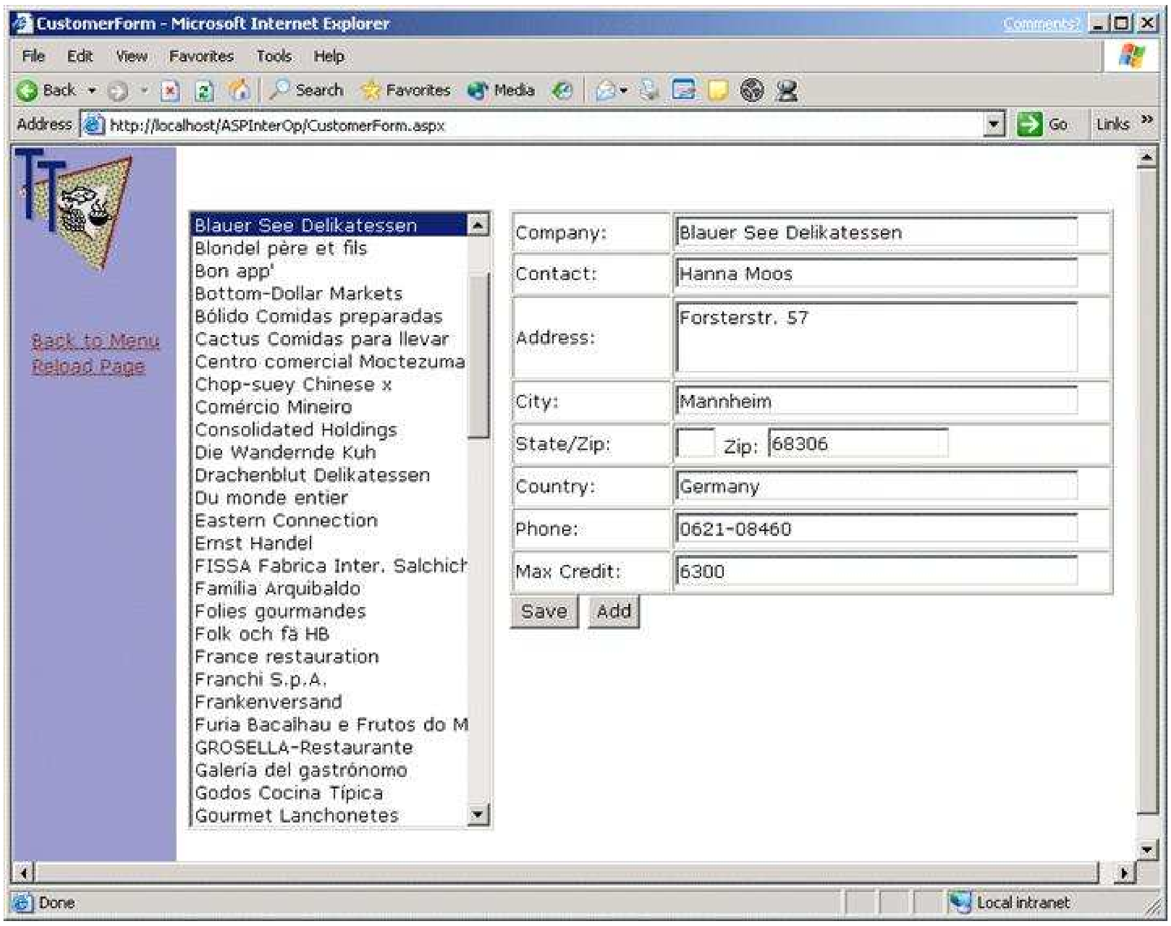
Note that you have to cast the data items to a specific type as the actual fields are objects. You can use ToString() on strings, but you'll need to use casts for most other datatypes.

Take 2: A little more generic

The code above is fine, but it gets repetitive in a hurry, so rather than write it each time it's a good idea to isolate some of the data access code into a separate business object layer. This is overly simplistic for this article's purposes, but demonstrates the basics of what it takes to simplify data access and separate the business logic from the ASP.NET page.

The following example creates a small ASP.NET application that displays a list of customers and lets you browse through the list and view and edit each of the customers. You can also add new customers. I'll demonstrate querying, updating and finally adding a customer using a stored procedure in this example all using the simple business object template.

To do this I use a very simplified business object called tasCustomer that's subclassed from aBusObj, which is a very minimal helper business object class that simplifies some of the queries and update operations.


CustomerForm - Microsoft Internet Explorer
Comments?
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/ASPInterOp/CustomerForm.aspx
Go Links
Back to Menu
Reload Page
Save Add
Done
Local intranet

Blauer See Delikatessen
Blondel père et fils
Bon app'
Bottom-Dollar Markets
Bólido Comidas preparadas
Cactus Comidas para llevar
Centro comercial Moctezuma
Chop-suey Chinese x
Comércio Mineiro
Consolidated Holdings
Die Wandernde Kuh
Drachenblut Delikatessen
Du monde entier
Eastern Connection
Ernst Handel
FISSA Fabrica Inter. Salchich
Familia Arquibaldo
Folies gourmandes
Folk och fä HB
France restauration
Franchi S.p.A.
Frankenversand
Furia Bacalhau e Frutos do M
GROSELLA-Restaurante
Galería del gastrónomo
Godos Cocina Típica
Gourmet Lanchonetes

Company:	Blauer See Delikatessen
Contact:	Hanna Moos
Address:	Forsterstr. 57
City:	Mannheim
State/Zip:	Zip: 68306
Country:	Germany
Phone:	0621-08460
Max Credit:	6300

Figure 1 *– This customer form browses and allows editing of customers from the TasTrade sample application.*

This Web Form performs two separate data tasks: Retrieval of the list displayed and the update operations on the individual customers.

Since this is ASP.NET I'll use DataSets for these operations and also take advantage of the fact that the form's viewstate can persist the list for me without going back to the database. In essence the form loads the list of customers only once, and then persists the list as part of the form's viewstate that is passed back and forth over the wire. This view state also includes settings of the list such as the currently selected item(s). This works well for smallish lists like this one, but I wouldn't recommend this same approach for long lists as you are essentially trading bandwidth for server resources against a database operation. If you don't use ViewState you have to manually retrieve the data each time for the list, and then set the selected item based on the form variable returned from the list (Request.Form("oCustomerList")).

Ok, let's see how we retrieve this data using the VFP OleDb driver. As I mentioned I'm using a simple business object implementation to help with my work here. In the ASP.NET form (CustomerForm.aspx) the first thing I do is run an Execute to return the list if the form is not in PostBack mode. PostBack mode for an ASP.NET form means that the form is not being loaded for the first time and returning values from the form – in other words the user has clicked a button. The first time through the form will not be in PostBack mode and we'll have to run the query. To do so I use in the form's Page_Load event method (Listing 2 from CustomerForm.aspx.cs).

Listing 2 (C# ASP.NET): Retrieving and displaying the customer list and first record

```
private void Page_Load(object sender, System.EventArgs e)
{
   // *** Clear the Error Message always
   this.lblErrorMessage.Text = "";

   this.oCustomer = new tasCustomer();

   if (!this.IsPostBack)
   {
      if (this.LoadCustomerList()) {
         // *** Force first item to display in edit view
         if (this.oCustomerList.Rows > 0)
         {
            this.oCustomerList.SelectedIndex = 0;
            this.ShowCustomer();
         }
      }
   }
}

protected bool LoadCustomerList()
{
   int lnCount = this.oCustomer.Execute(
                     "select company, cust_id from customer order by Company",
                     "customer");

   if (this.oCustomer.lError)
   {
      this.lblErrorMessage.Text = this.oCustomer.cErrormsg;
      return false;
   }
   else
   {
      // *** Databind the DataTable
      this.oCustomerList.DataSource = this.oCustomer.oDS.Tables["customer"];
      this.oCustomerList.DataTextField = "company";
      this.oCustomerList.DataValueField = "cust_id";
      this.oCustomerList.DataBind();
   }

   return true;
}
```

This code fires off in the Page_Load of the ASP.NET page and then calls the LoadCustomerList() method to actually perform the data retrieval and binding of data tied to the list. The Execute method on the business object handles running the query. The business object provides several methods such as Open() and Execute() that remove some of the code that is required in setting up a DataSet and returns the actual DataSet as an object member oDS. oCustomer.oDS holds the result and as you can see in the code above the "customer" result table is then bound to the list box. Both a display field and a key field are bound – we'll use the key field later on to retrieve a specific customer.

The key methods of the base business object are set up as shown in Listing 3 (from tasCustomer.cs).

Listing 3 (C#): Simplified base business object for handling data retrieval

```
public class aBusObj
{
   public string cConnectString =   "";

   /// these properties along with Open,Close,Load,Save really belong
   /// into a data wrapper or high level business object. For compactness'
   /// sake I've included them as part of the class here.
   public OleDbConnection oConn         =   null;
   public DataSet   oDS                 =   null;
   public string cErrormsg              =   "";
   public bool lError                 =   false;

protected virtual bool Open()
{
   /// create if it doesn't exist already
   if (this.oConn == null)
   {
      try
      {
         this.oConn = new OleDbConnection(this.cConnectString);
      }
      catch(Exception e)
      {
         this.SetError( e.Message );
         return false;
      }
   }

   /// check if connection is open - if not open it
   if (this.oConn.State != ConnectionState.Open)
   {
      try
      {
         oConn.Open();
      }
      catch(Exception e)
      {
         this.SetError( e.Message );
         return false;
      }
   }

   /// make sure our dataset object exists
   if (oDS == null)
```

```
      oDS = new DataSet();

   return true;
}

protected virtual bool Close()
{
   if (oConn.State == ConnectionState.Open)
      try
      {
         oConn.Close();
      }
      catch(Exception e)
      {
         this.SetError( e.Message );
         return false;
      }

   return true;
}

public virtual int Execute(string lcSQL, string lcCursor)
{
   if (!this.Open())
      return 0;

   OleDbDataAdapter oDSAdapter = new OleDbDataAdapter();
   oDSAdapter.SelectCommand = new OleDbCommand(lcSQL,oConn);

   // *** remove the table if it exists - fail and ignore
   try
   {
      oDS.Tables.Remove(lcCursor);
   }
   catch(Exception e) { }  // Ignore the error

   try
   {
      oDSAdapter.Fill(oDS,lcCursor);
   }
   catch(Exception e)
   {
      this.SetError(e.Message);
      return 0;
   }

   return oDS.Tables[lcCursor].Rows.Count;
}

// … more methods here not included – see source code

} // end of aBusObj
```

The Open() and Close() metods basically deal with setting up the connection and making sure that the connection strings are valid. The Execute() method then runs the specified query and creates a cursor in the business object's DataSet object. If you'll recall a DataSet can contain more than a single table, so if you run multiple queries they can all be stored in a single dataset. If you have multiple business objects they can also potentially share the same DataSet object reference. Note that Execute tries to remove the table name before creating the result so that if a cursor by the same name exists already it's overwritten – otherwise an exception would occur.

With this code in place running a query then becomes as easy as calling the Execute method with a SQL statement. In this case the SQL statement retrieves the customer list, and then assigns the result table to the listbox of the TasTrade customer form.

Note that all the methods in the aBusObj class are virtual, which means they can be overridden by the implementation classes. For example, the Open method is overridden in the tasCustomer class to set properties on the VFP OleDb session. In order to do this you have to actually run a command against the VFP data source using language commands such as SET EXCLUSIVE OFF, SET DELETED ON etc (Listing 3.1 from tasCustomer.cs).

Listing 3.1 (C#): Overridden Open() method that handles VFP OleDb environment settings

```
protected override bool Open()
{
   if (base.Open())  // call the Parent class' Open() (no laughing please! <g>)
   {
      try
      {
         OleDbCommand oCommand = new OleDbCommand();
         oCommand.Connection = this.oConn;
```

```
            oCommand.CommandText = "SET NULL OFF\r\nSET DELETED ON";
            oCommand.ExecuteNonQuery();
         }
         catch(Exception ex)
         {
            this.SetError(ex.Message);
            return false;
         }

         return true;
      }

      return false;
   }
```

Databinding record data fields

Once the list has been loaded the next step is to retrieve an individual customer for display. To accomplish this task I use ASP.NET databinding, which is somewhat limited if you compare it to databinding in desktop applications. Keep in mind ASP.NET databinding is really a one way affair meant only to bind data for display – to read data back you have to explicitly extract the data from the controls that it's held in.

To do the databinding I use a combination of the Server control expressions (<%# %> syntax) and code in the ShowCustomer method. ShowCustomer() get fired from the form's SelectedIndex_Changed event method (Listing 4 from CustomerForm.aspx.cs).

Listing 4 (C#): Displaying a by binding the individual fields to a DataRow

```
private void oCustomerList_SelectedIndexChanged(object sender,EventArgs e)
{
   this.ShowCustomer();
}

protected string GetIdFromList()
{
   if (this.oCustomerList.SelectedItem == null)
   {
      this.lblErrorMessage.Text = oCustomer.cErrormsg;
      return null;
   }

   return this.oCustomerList.SelectedItem.Value;
}

private bool ShowCustomer()
{
   // *** Figure out which item is selected
   string lcID = this.GetIdFromList();

   // *** Load that customer
   if (!this.oCustomer.LoadCustomer(lcID)) {

      this.lblErrorMessage.Text = oCustomer.cErrormsg;
      return false;
   }

   // *** Force databinding to occur to the ASP Server Controls
   this.dtrCurrent  = this.oCustomer.oDS.Tables["Customer"].Rows[0];
   this.DataBind();

   return true;
}
```

The business object LoadCustomer method is fired which does little more than run an Execute and retrieve the single record using the existing business object (Listing 5 from tasCustomer.cs).

Listing 5 (C#): Retrieving a single customer with the business object

```
public bool LoadCustomer(string lcID)
{

   int lnCount = this.Execute("select * from customer where cust_id='" +
                              lcID + "'","Customer");
   if (this.lError)
   {
      return false;
   }

   return true;
}
```

This single record table row is then assigned to the dtrCurrent property of the form, which in turn is databound to the ASP.NET server controls.

DataBinding in .NET to textbox controls is one way which means that data is bound for display only. This makes sense given the stateless nature of Web requests and display and update happening usually in the same request.

Databinding to text box controls is accomplished by setting the Value tag of the ASP.NET control in the ASP.NET display page. For example the Company field is bound as follows in CustomerForm.aspx:

```
<asp:TextBox id="txtCompany" runat="server"
        Text='<%# this.dtrCurrent["company"] %>'>
</asp:TextBox>
```

Note that databinding is not automatic – it doesn't occur until you explicitly call the DataBind method of the individual control or the Page object. Above the ShowCustomer() method calls the DataBind() method of the page (this.DataBind())to force all the controls to bind. This is good too, because of the way that the form works. First the list loads, and no items are available yet. An item is loaded only after the list is loaded and the first item can be selected.

This is important to understand as you can't bind to a datasource that is not loaded yet or else the page will fail. ASP.NET gives you full control on when the actual data is bound while allowing you to set the binding expression any time. Hey, that's a request I've had for VFP forms forever and it actually works here.

Updating the data

As I mentioned above ASP.NET data is not really databinding in the traditional sense because it's only one way. So in order to update data you have to read the values explicitly from the controls and back into the underlying data source. Here I read the data back into the DataRow object after pressing the save button (Listing 6 from CustomerForm.aspx.cs).

Listing 6 (C#): Saving a customer record

```
private void btnSave_Click(object sender, System.EventArgs e)
{
   string lcID = this.GetIdFromList();

   if   ( !this.oCustomer.LoadCustomer(lcID) )
   {
      this.lblErrorMessage.Text = oCustomer.cErrormsg;
      return;
   }

   /// Update the row contents
   DataRow oRow  = this.oCustomer.oDS.Tables["Customer"].Rows[0];
   oRow["company"] = this.txtCompany.Text;
   oRow["address"] = this.txtAddress.Text;
   oRow["city"] = this.txtCity.Text;
   oRow["MaxOrdAmt"] =  Convert.ToDecimal(this.txtCredit.Text);
   // … more fields omitted here

   /// Call the bus object to save the data to disk
   this.oCustomer.SaveCustomer();
}
```

The code then defers to the business object to save the data back to the VFP data source.

Up to now our VFP datasource has had no issues in interaction with ADO.NET. Unfortunately for any kind of data updates the VFP OleDb provider has a few missing features that make an otherwise simple process a lot more code intensive than it is say with a SQL Server datasource.

ADO.NET supports the concept of database schemas being retrieved along with the data when a query is run. So, normally when you run a query against a datasource the Primary key info is retrieved. This makes it possible to automatically update a datasource because the DataAdapter can use the schema information to decide how to update the DataSource based on the key info.

Unfortunately the VFP OleDb provider in its current incarnation does not support this information and you're required to manually create INSERT and UPDATE statements. The following work with SQL Server, but **not** with VFP. With SQL Server you can do the following:

```
OleDbDataAdapter oDSAdapter = new OleDbDataAdapter("select * from customer",
                                                   this.oConn);

/// This builds the Update/InsertCommands for the data adapter
OleDbCommandBuilder oCmdBuilder = new OleDbCommandBuilder(oDSAdapter);

lnRows = oDSAdapter.Update(this.oDS,"Customer");
```

The OleDbCommandBuilder is an object that can automatically build SQL INSERT/UPDATE/DELETE commands if the datasource supports schema information. But this doesn't work with the VFP OleDb driver as this information is not provided.

To work around this you have to manually assign the INSERT/UPDATE/DELETE statements. The following example demonstrates the UPDATE command (limited to a couple of fields).

```
string lcSQL =
   "UPDATE customer SET company='" + oRow["company"].ToString() +  "' " +
   "where cust_id='" +oRow["cust_id"]+ "'";

oDSAdapter.UpdateCommand = new OleDbCommand(lcSQL,this.oConn);
```

Once that's done the oDSAdapter.Update() method call succeeds.

Of course if you have to go through all the trouble of updating the UPDATE command manually you might as well just execute the UPDATE command manually with less code without requiring the DataSet to be directly involved at all:

```
string lcSQL =
     "UPDATE customer SET company='" + oRow["company"].ToString() +  "' " +
     "where cust_id='" +oRow["cust_id"]+ "'";

OleDbCommand oCommand = new OleDbCommand(lcSQL,this.oConn);
oCommand.ExecuteNonQuery();
```

Schema retrieval functionality is scheduled to be provided in the next version of the VFP OleDb provider, but currently we're stuck with having to manually write our own UPDATE/INSERT statements. Putting all of this together the SaveCustomer method looks like Listing 6.1 (tasCustomer.cs).

Listing 6.1 (C#): Saving a customer record from the DataSet back to the VFP table

```
public bool SaveCustomer()  // tasCustomer object
{
   if (!this.Open())
   {
      return false;
   }

   OleDbDataAdapter oDSAdapter = new
            OleDbDataAdapter("select * from customer",this.oConn);

   // This builds the Update/InsertCommands for the data adapter
   // THIS DOES NOT WORK WITH VFP DATA!
   // OleDbCommandBuilder oCmdBuilder = new OleDbCommandBuilder(oDSAdapter);

   DataRow oRow = this.oDS.Tables["Customer"].Rows[0];

   string lcSQL =
      "UPDATE customer " +
      "SET company='" + oRow["company"].ToString() + "'," +
          "contact='" + oRow["contact"].ToString() + "'," +
          "address='" + oRow["address"].ToString() + "'," +
          "city='" + oRow["city"].ToString() + "', " +
          "region='" + oRow["region"].ToString() + "', " +
          "postalcode='" + oRow["postalcode"].ToString() + "'," +
          "country='" + oRow["country"].ToString() + "'," +
          "maxordamt=" + oRow["maxordamt"].ToString() + " " +
      "where cust_id='" +oRow["cust_id"]+ "'";

   oDSAdapter.UpdateCommand =
       new OleDbCommand(lcSQL,this.oConn);

   int lnRows = 0;
   try
   {
      /// Take the changes in the dataset and sync to the database
      lnRows = oDSAdapter.Update(this.oDS,"Customer");

     // *** Or you can just directly execute the Command object
     // oDSAdapter.UpdateCommand.ExecuteNonQuery();
   }
   catch(Exception e)
   {
      this.cErrormsg = e.Message;
      return false;
   }

   return true;
}
```

Using Stored Procedures

As a last issue here I want to demonstrate calling a Stored Procedure in a VFP DBC to perform the Insert operation for a new customer. Again, standard syntax that's supported with SQL Server doesn't work with the VFP OleDb provider at this time, so you need to use explicit syntax to call the stored procedure.

I kept this method to only a few field values to keep these code snippets short enough for display – you'd probably want to add all the remaining fields. The VFP Stored Procedure looks like this added to the TasTrade sample data (testdata.dbc):

Listing 6.5 (VFP): Insert Customer VFP Stored Procedure

```
FUNCTION InsertCustomer
LPARAMETERS lcCompany, lcContact, lnMaxOrdAmt

lcCustId = LEFT( UPPER(lcCompany), 5)
INSERT INTO Customer (CUST_ID,Company, Contact, MaxOrdAmt)
  VALUES (lcCustId,lcCompany,lcContact,lnMaxOrdAmt)

RETURN lcCustId
```

When you click on the Add button of the ASP.NET form the currently active data is stored to the database by calling the AddCustomer method of the business object.

```
private void cmdAdd_Click(object sender, System.EventArgs e)
{
 if (!this.oCustomer.AddCustomer(this.txtCompany.Text,this.txtContact.Text,
                                 Convert.ToDecimal( this.txtCredit.Text)) )
   {
      this.lblErrorMessage.Text = this.oCustomer.cErrormsg;
   }
}
```

AddCustomer in the business object then calls the stored procedure explicitly (Listing 7 – tasCustomer.cs).

Listing 7 (C# ADO.NET): Adding a customer with a stored procedure

```
public bool AddCustomer(string lcCompany, string lcContact, decimal lnMaxOrdAmt)
{

if (!this.Open())
{
   return false;
}

OleDbCommand oCommand = new OleDbCommand();
oCommand.Connection = this.oConn;
oCommand.CommandText = "InsertCustomer('" + lcCompany + "','" +
                                        lcContact + "',lnMaxOrdAmt.ToString())";

try
{
   oCommand.ExecuteNonQuery();
}
catch(Exception ex)
{
   this.SetError(ex.Message);
   return false;
}

return true;
}
```

I ran into some problems here with this code that worked fine in VFP, but didn't at first with .NET due to the NULL requirements. The OleDb provider string doesn't support provider options so the only way to set these particular settings is to actually run a command against the connection. To do this I overrode the Open() method of the aBusObj class to do the following (tasCustomer.cs):

Listing 8 (C#): Setting Language Options for the current OleDb session

```
protected override bool Open()
{
   if (base.Open())
   {
      OleDbCommand oCommand = new OleDbCommand();
      oCommand.Connection = this.oConn;
      oCommand.CommandText = "SET NULL OFF\r\nSET DELETED ON";
      oCommand.ExecuteNonQuery();
      return true;
   }

   return false;
}
```

Interestingly enough I also ran into a major bug in the driver before I added the code above. The INSERT from the stored procedure would fail with an error: *Field TITLE cannot contain Nulls.* The Command's ExecuteNonQuery() would throw an exception indicating that the stored procedure call failed, yet when I checked the table I ended up with a new record anyway! This issue appears to be specific to this NULL error as other errors like violating the Primary key rule did not cause this behavior. Microsoft is aware of this issue and looking into it.

OleDb Summary

The VFP OleDb provider makes it possible to access Visual FoxPro data through the .NET environment but in its current implementation it doesn't provide all the functionality that ADO.NET offers. This doesn't exclude VFP from .NET integration, but it does make it more code intensive to use VFP data than say SQL Server or even Access (Jet) data. Plan on writing SQL statements by hand as opposed to using ADO.NET's auto-update features at least in this release of the driver. Microsoft has indicated that they are planning to support more of the .NET specific features in future versions of the driver – they are aware of some of the limitations and are working to address them at this time so we can look forward to fuller .NET support.

COM Interoperability

For many years the main call from Microsoft has been to build component based applications based on the Component Object Model (COM) by creating components in your language of choice and then exposing those components or classes as COM objects to the operating system.

.NET bucks this trend by using a whole new mechanism of interoperability via the Common Language Runtime which provides intermediary byte code that is compatible with multiple languages that can output .NET capable byte code that is compiled into executable binaries on the fly by the runtime. What this means is that the days for binary incompatibility are gone. As are some of the issues that have plagued the COM infrastructure – namely the issue of versioning and DLL Hell.

However, COM remains very important even with .NET both as a backwards compatibility feature as well as a mechanism to provide interop with the runtime as a whole. For example, many of the scalability features of the Windows operating system are still implemented using the COM+ system. .NET itself also uses COM+ underneath to implement many of the Enterprise features like distributed transaction management, context and process management and many security features. What's exciting here is though that the intricacies of COM+ are mostly hidden behind more user friendly .NET classes that expose the functionality through class interfaces or attribute based descriptors.

However, COM usage in .NET for the developer is played way down and the focus in .NET is on building .NET classes that perform functionality rather than calling out to 'legacy' COM components to perform business logic or other tasks. COM interop is provided more as a backward compatibility feature than a 'moving forward' feature. But because there's a large, large investment in COM by many organizations, COM support in .NET is fairly strong.

The good news is that you can access most COM components in .NET without any fuss. It isn't quite as easy as it was say in Visual FoxPro/Visual Basic with late binding and CreateObject(), but with a little bit more work COM interop is easily accomplished. The not so good news though is that there's a performance penalty for using COM objects in .NET resulting in reduced performance when compared to previous COM based technologies. It's clear that the focus of .NET is not necessarily to provide the best hosting environment to COM components but rather to provide a mechanism to allow applications to co-exist until they can be rebuilt with .NET.

COM interop in .NET is a two way affair – you can call COM components from .NET and you can call .NET components from classic COM capable applications. Let's look at each of the approaches, how they work and what they offer to the developer.

Calling COM components from .NET

This mechanism is likely to be the most commonly used functionality of COM interop as it allows you to integrate existing functionality into new applications that are being built or migrated to the .NET platform.

This process uses internal Interop classes in the .NET framework to link and bind to COM components. The standard process to accomplish this is to import the COM component which creates a compiled wrapper .NET class that implements the COM objects interfaces as a .NET class.

Because .NET is a type safe environment this .NET interface is necessary to allow access to the class methods and properties. This satisfies the compiler for type safety through mapping all the COM properties/parameters/return types to the appropriate .NET types. This process is similar to early binding in traditional COM client, but yet it is somewhat different. Behind the scenes the wrapper class makes the appropriate Interop class calls to call the COM component and then maps the parameters and return values to the wrapper class.

Those Interop classes can also be accessed directly via code to provide the equivalent of Late Binding, but be advised that this can take a lot of code to do. It's similar to using C++ and the IDispatch COM interface to query information about a class and then calling the method signature with generic parameters that need to be translated and typecast into the proper typesafe values that the compiler can deal with. This code can be messy and I wouldn't recommend it for anything but objects that aren't properly described inside of their respective type libraries (more on this later).

To demonstrate let's create a simple object that I've used in the past for ASP COM interop and walk through creating the object in VFP and then making it available in .NET to an ASP.NET page/application. The class is very simple and doesn't do anything fancy – the purpose here is to demonstrate the operation of the interop mechanism not to show what you can do with it. I'll leave that for some future article <g>.

I'll create a COM object here and show a few methods at a time. To review the basics of creating a COM object, I'll start with a simple object that serves as a Counter manager that stores and increases named counter values. The first step is to create the object as a COM capable class (listing 9 – asptools.prg).

Listing 9 (VFP): A simple Counter VFP COM object

```
*************************************************************
DEFINE CLASS ASPTools AS Custom OLEPUBLIC
```

```
*************************************************************

*** Custom Properties
cDataPath=LOGFILEPATH
cAppStartPath = ""
oScriptingContext = .NULL.

nCounter = 0

lError = .f.
cErrorMsg = ""

************************************************************************
* aspTools :: Init
*********************************
***  Function: Set the server's environment. IMPORTANT!
************************************************************************
FUNCTION INIT

*** Make all required environment settings here
*** KEEP IT SIMPLE: Remember your object is created
***                 on EVERY ASP page hit!
SET RESOURCE OFF   && Best to compile into a CONFIG.FPW
SET EXCLUSIVE OFF
SET REPROCESS TO 2 SECONDS

SET CPDIALOG OFF
SET DELETED ON
SET EXACT OFF
SET SAFETY OFF

*** Add this mainly so that the project will include
*** this stuff here
SET PROCEDURE TO wwUtils ADDITIVE
SET PROCEDURE TO wwAPI ADDITIVE

*** IMPORTANT: Figure out your DLL startup path
THIS.cAppStartPath = ADDBS(JUSTPATH(Application.ServerName))

*** If you access VFP data you probably will have to
*** use this path plus a relative path to get to it!
*** You can SET PATH here, or else always access data
*** with the explicit path
SET PATH TO (THIS.cAppStartpath)
DO PATH WITH THIS.cAppStartPath + "DATA"

ENDFUNC


************************************************************************
* aspTools :: IncCounter
*********************************
***  Function: Increments a counter in the registry.
***      Pass: lcCounter  -  Name of counter to increase
***            lnValue    -  (optional) Set the value of the counter
***                          -1 delete the counter.
***    Return: Increased  Counter value  -  -1 on failure
************************************************************************
FUNCTION IncCounter(lcCounter as String, lnSetValue as Integer) as Integer
LOCAL lnValue

lnSetValue=IIF(EMPTY(lnSetValue),0,lnSetValue)

oMTS  = CreateObject("MTxAS.AppServer.1")
THIS.oScriptingContext = oMTS.GetObjectContext()
this.oScriptingContext.Item("Response").Write("Inccounter from VFP<p>")

IF !USED("WebCounters")
   IF !FILE(THIS.cAppStartPath + "WebCounters.dbf")
        SELE 0
          CREATE table (THIS.cAppStartPath + "WEBCOUNTERS") ;
          (    NAME        C (20),;
               COUNTER       I )
           USE
   ENDIF
   USE (THIS.cAppStartPath + "WEBCOUNTERS") IN 0 ALIAS WebCounters
ENDIF

SELE WebCounters

LOCATE FOR UPPER(name) = UPPER(lcCounter)
IF !FOUND()
```

```
      INSERT INTO WEBCounters  VALUES (lcCounter,1)
      lnValue = 1
   ELSE
      IF RLOCK()
         IF lnSetValue > 0
            REPLACE Counter with lnSetValue
         ELSE
            IF lnSetValue < 0
               REPLACE Counter with 0
               DELETE
            ELSE
               REPLACE Counter with Counter + 1
            ENDIF
         ENDIF
         lnValue = Counter
         UNLOCK
      ELSE
         lnValue = 0
      ENDIF
   ENDIF

   *** For testing
   This.ncounter = lnValue

   RETURN lnValue
   ENDFUNC

   ENDDEFINE
```

Remember that's it's real easy to create a COM object in VFP simply by setting the OLEPUBLIC attribute on the DEFINE CLASS statement of a PRG based class or by setting the OLE Public checkbox on the class property settings of a VCX based class. However, it's equally important to set up your COM object properly to run in the hosted environment, especially if that hosted environment happens to be a service such as Internet Information Server (IIS). The INIT method of the object sets the environment and saves some state information that preserves the startup location of the application and set's the path to it so we can find the data located there.

To create this class as a COM object, add this class to a project (I named mine ASPDemos) and COMPILE it into a Multi-Threaded COM Server or use BUILD MTDLL AspDemos from AspDemos.

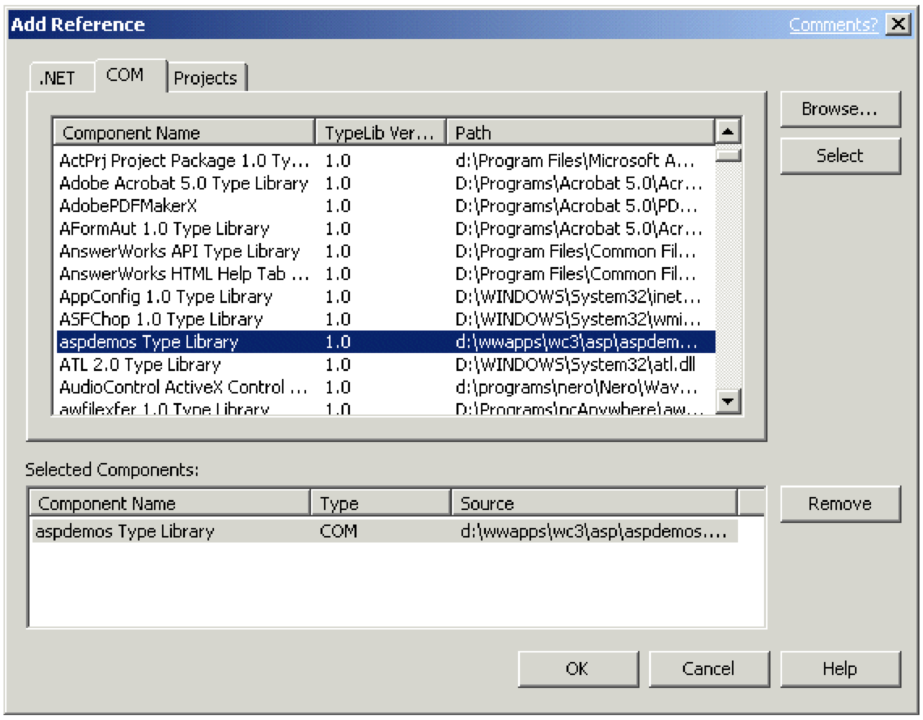
Once this is done you can now use the object from VFP with:

```
oDemo = CREATEOBJECT("AspDemos.AspTools")
? o.IncCounter("CodeDemo")  && 1
? o.IncCounter("CodeDemo")  && 2
? o.IncCounter("EssentialFoxDemo")  && 1
? o.IncCounter("CodeDemo")  && 3
```

For most of you this is nothing new. But note that if you are building COM objects for use with .NET there is at least one major difference: You MUST use VFP 7.0 and it's typing features to describe your parameter and return value types, or else your methods will not be accessible properly.

Importing the COM object into .NET

Ok, once your object is built you need to import it into .NET. The easiest way to do this is to use Visual Studio.NET and the References selection in the Project manager. Right click on the References tree item and Add Reference. Then pick the COM tab and select your object from the list, or if it's not registered yet, browse to it on disk (Figure 2).


Add Reference

Comments?

.NET | COM | Projects

Component Name	TypeLib Ver...	Path
ActPrj Project Package 1.0 Ty...	1.0	d:\Program Files\Microsoft A...
Adobe Acrobat 5.0 Type Library	1.0	D:\Programs\Acrobat 5.0\Acr...
AdobePDFMakerX	1.0	D:\Programs\Acrobat 5.0\PD...
AFormAut 1.0 Type Library	1.0	D:\Programs\Acrobat 5.0\Acr...
AnswerWorks API Type Library	1.0	D:\Program Files\Common Fil...
AnswerWorks HTML Help Tab ...	1.0	D:\Program Files\Common Fil...
AppConfig 1.0 Type Library	1.0	D:\WINDOWS\System32\inet...
ASFChop 1.0 Type Library	1.0	D:\WINDOWS\System32\wmi...
aspdemos Type Library	1.0	d:\wwapps\wc3\asp\aspdem...
ATL 2.0 Type Library	1.0	D:\WINDOWS\System32\atl.dll
AudioControl ActiveX Control ...	1.0	d:\programs\nero\Nero\Wav...
awfilexfer 1.0 Type Library	1.0	D:\Programs\pcAnywhere\aw...

Browse...

Select

Selected Components:

Component Name	Type	Source
aspdemos Type Library	COM	d:\wwapps\wc3\asp\aspdemos....

Remove

OK | Cancel | Help

***Figure 2** – Importing a COM object into .NET is as easy as selecting the object from the installed COM components or physically selecting a typelibrary/dll from disk. This process creates a .NET wrapper class for your COM object that is treated as any other .NET object.*

Once the object has been added to your project you can access it pretty much like any other .NET object. For ASP.NET pages you'll have to do a couple of additional things to get it to run properly. Since I'm using VS.NET the actual ASP.NET page contains nothing more than the page header, while all the logic goes into a codebehind class. The main thing on the ASPX page header is the PAGE directive:

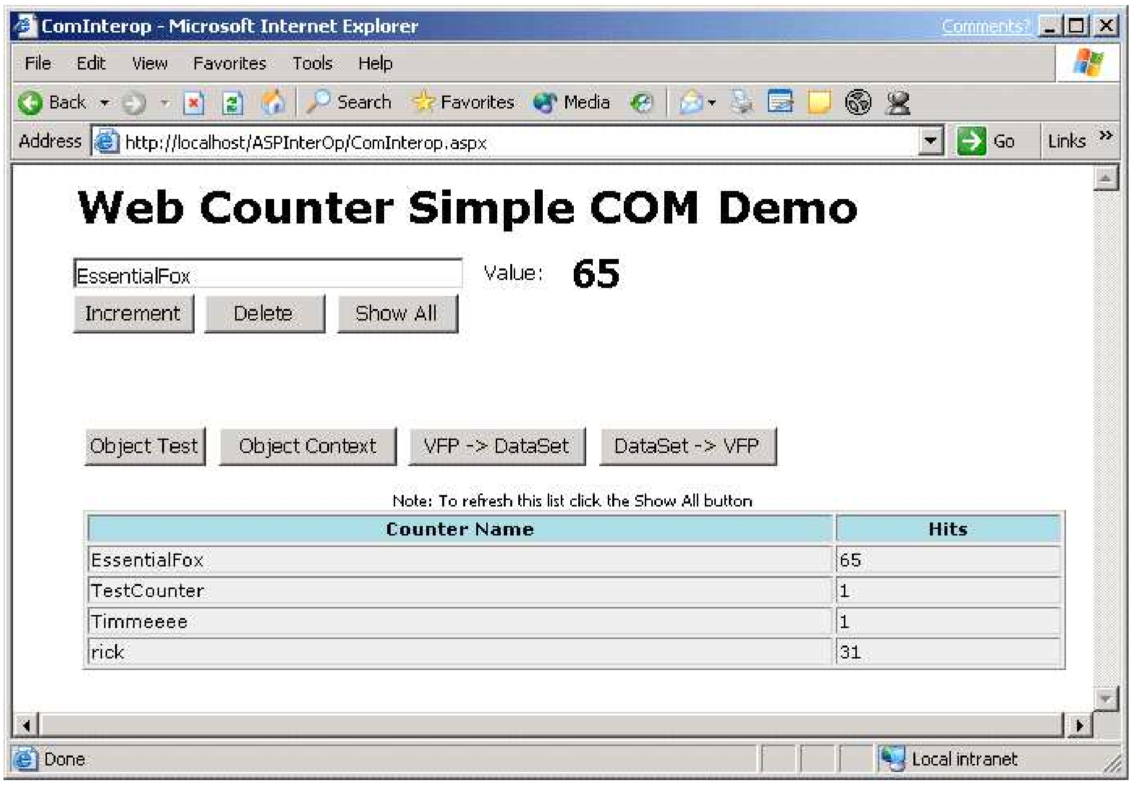
```
<%@ Page ASPCOMPAT=true language="c#" Codebehind="ComInterop.aspx.cs"
AutoEventWireup="false" Inherits="ASPInterOp.ComInterop" %>
```

This tells the CLR that the actual code that runs on this page is stored in COMInterop.aspx.cs. This is where I'll put the code to demonstrate the use of the COM object I created above.

There's another very important directive in the PAGE tag above! The ASPCOMPAT=true attribute tells the ASP.NET page that it is hosting COM components that don't conform to the ASP.NET threading model. Visual FoxPro (and Visual Basic 6) COM objects are Single Threaded Apartment (STA) objects, while .NET natively runs ASP.NET requests on MultiThreaded Appartment (MTA) threads. This requires the .NET framework to make some changes into how requests are handled by essentially switching the request into STA compatible mode. I'll talk more about this a little later in relation to how this affects performance.

ASPCOMPAT also provides compatibility to classic the ASP COM object environment by providing the appropriate Context objects that were supported to retrieve the ASP intrinsic objects before. In particular the ObjectContext object provider which uses the COM+ IApplicationContext object interface is provided to COM objects which would otherwise not be available. I'll come back to this in a bit as well.

The code behind page then contains all the logic of the page with the ASPX page only containing the display elements. I want to keep things simple so I created a simple form that lets you increment counters, delete them and show them all. The final running form is shown in Figure 3.


ComInterop - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/ASPInterOp/ComInterop.aspx
Go Links
Done
Local intranet

Web Counter Simple COM Demo

EssentialFox Value: 65

Increment Delete Show All

Object Test Object Context VFP -> DataSet DataSet -> VFP

Note: To refresh this list click the Show All button

Counter Name	Hits
EssentialFox	65
TestCounter	1
Timmeeee	1
rick	31

***Figure 3** – The COM Counter sample demonstrates using a basic COM object in a Web form.*

The actual page is made up of a textbox (txtCounterName), a label (lblCounter), a set of operational buttons (btnIncrement, btnDelete, btnShowAll) and an empty label control that is filled from the COM object – in this case with an HTML string that shows the table you see in Figure 3. There's also a label between the table and the buttons called lblErrorMsg that is used for status information. For example, when you delete a counter this label is updated with a message that says that the counter was deleted.

Let's start with how the basic incrementation mechanism works. Here's the Page Load and Increment button click code (Listing 10 – ComInterop.aspx.cs).

Listing 10 (C# ASP.NET): Simple ASP.NET Codebehind class

```
...
using aspdemos;  // COM object wrapper

namespace ASPInterOp
{
   // … control declarations left out

   public class ComInterop : System.Web.UI.Page
   {
      private void Page_Load(object sender, System.EventArgs e)
      {
         if (!this.IsPostBack)
         {
            this.btnIncrement_Click(this,e);
         }
      }

      private void btnIncrement_Click(object sender, System.EventArgs e)
      {
         ASPToolsClass oVFP = new ASPToolsClass();
         this.lblCounter.Text =
              oVFP.IncCounter(this.txtCounterName.Text,0).ToString();
      }
}
```

Ok, let's look at what happens here. First make sure the aspdemos namespace which represents the imported COM object, is added to your class so that you can access the class. With the namespace and reference added our VFP class behaves just like any other .NET class including full access to Intellisense (Figure 4).

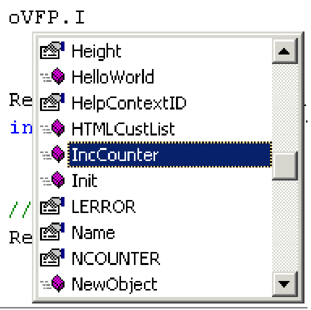


Figure4 *– Once imported your COM object looks like any other .NET object with full access to Intellisense.*

It's important that you define your methods with proper Type information using VFP 7.0's parameter typing.

```
FUNCTION IncCounter(lcCounter as String, lnSetValue as Integer) as Integer
```

If you don't provide the type information, the method will not import properly and show up without parameters and return values – and will fail at runtime when called. (Note: as of .NET Framework SP1 this has been fixed – methods with variants will import with *object* parameters, which still are a pain to use so make sure you use typed parameters/returns whenever possible (ie. all the time!!!)).

Properties names are translated to upper case so nCounter becomes NCOUNTER. Remember C# is case-sensitive so this makes a big difference. Better yet, always use Intellisense to provide you with the proper property names.

Note that all properties are returned as type Object unless you specifically type them with VFP's _ComAttrib functionality part of the class definition. Type object is similar but not quite the same as a Variant in COM. Object types can contain different sub types (such as strings, ints etc), but they are cannot directly be accessed by their sub-type. In order to use the variable safely in C# you have to explicitly cast it in many cases:

```
          int lnCounter = (int) oVFP.NCOUNTER + 10;
          this.txtCounterProperty.Text = lnCounter.ToString();
```

If you use the object variable multiple times it's probably more efficient to store it into a properly typed variable first, then use it as you intend to. It's both easier to work with and faster in execution.

Changing Code in your COM object

When it comes time to make a change to the COM object, things are very similar to the way it was with classic ASP – in order to compile the COM object you need to restart the Web Service or at least the Web application that hosts the COM component.

The easiest way to do this for me is to run the IISRESTART utility. And for this purpose I use an Intellisense script (tied to the *IIS* key sequence) that simply does:

```
RUN /n4 IISRESET
```

IISRESET is a IIS 5.0 and later utility that has a number of options for restarting the Web server. It can restart all the services or with the proper options any of the specific services. By the way, it can also be used to reboot a machine reliably. The nice thing about running IISRESET from Fox is that it runs in a separate window that starts and finishes when it's done so there's no more user interaction required.

While the reset is working let's add another method to our server to delete a counter (Listing 11 - asptools.prg).

Listing 11 (VFP): Deleting a counter in the VFP COM code

```
Function DeleteCounter(lcCounterName as String) as Boolean

IF EMPTY(lcCounterName)
   RETURN .F.
ENDIF

DELETE FROM WebCounters where Name = lcCountername

RETURN .T.
```

Then rebuild your server with BUILD MTDLL AspDemos from ASPDemos. To hook up this method I'll edit the visual ASPX page add the btnDelete button and add the following event code to the click event:

Listing 12 (C# ASP.NET): Calling the Delete code from ASP.NET code

```
private void btnDelete_Click(object sender, System.EventArgs e)
{
   ASPToolsClass oVFP = new ASPToolsClass();
   oVFP.DeleteCounter(this.txtCounterName.Text);
   this.lblErrorMsg.Text = "Counter " + txtCounterName.Text + " deleted...";
   this.lblCounter.Text = "";
}
```

Before you add this code you might want to quickly recompile the project (Ctrl-Shift-B) to force the AspDemos type library code change to reflect in the .NET class and in the Intellisense list. Once you recompile the DeleteCounter() method is available, otherwise it is not. This means that VS.NET rebuilds your COM wrapper every time you make a change to it, which is convenient. But realize that a compile is required to make this happen – changing the class and not recompiling in .NET will likely result in a runtime error or COM version conflict (depending on if and how the COM interface has changed).

This is different from classic ASP – the COM object in .NET is early bound which means it's bound at compile time through the wrapper class. Any changes to the COM object require a recompile, so making a change to the COM object always require two steps:

1. Change and recompile the VFP COM object
2. Recompile the .NET COM client application

Types of data to return

I think you will find that if you want to use ASP.NET efficiently, you probably want to use COM objects as business objects as opposed to an output generator. The two examples I've shown so far are - albeit very simple – business object methods that return business data directly back to your application.

You may also want to return display output back to the ASP page. For example, you may run a query and based on this query generate HTML in your VFP code. While I don't think this is the best of ideas for ASP.NET Web Form applications, you can do it very easily as the following code demonstrates.

Start by creating another method on your COM object that returns in this case the list of counters (Listing 13 – asptools.prg):

Listing 13 (VFP): Returning a list of counters in HTML format from the COM object
```
FUNCTION ShowCounters() as String

IF !USED("WebCounters")
   IF !FILE(THIS.cAppStartPath + "WebCounters.dbf")
        SELE 0
          CREATE table (THIS.cAppStartPath + "WEBCOUNTERS") ;
          (    NAME        C (20),;
                VALUE       I )
           USE
   ENDIF
   USE (THIS.cAppStartPath + "WEBCOUNTERS") IN 0 ALIAS WebCounters
ENDIF

SELECT name as Counter_Name, value as hits, ;
      [<a href="ShowCounters.asp?Action=Delete&ID=]+name+[">Delete</a>] as Action ;
   FROM WebCounters ;
   order by 1 INTO CURSOR TQuery

*** Render the cursor as an HTML table
RETURN THIS.ShowCursor()
```

This method runs a query and then calls a method in the sample called ShowCursor() which generically generates an HTML table from the data. This HTML string is then returned back to the client.

In the .NET form add a blank label control and call it lblCounterList. We then assign the control the HTML string that is returned from the ShowCounters() call to the COM object.

Listing 14 (C# ASP.NET): Assigning HTML content to a label control
```
private void btnShowAll_Click(object sender, System.EventArgs e)
{
   ASPToolsClass oVFP = new ASPToolsClass();
   this.lblCounterList.Text =
      "<small><center>Note: To refresh this list click the " +
      "Show All button</center></small>" +
      oVFP.ShowCounters();
}
```

Now, if you run this form you'll notice something interesting if you're not familiar with Web forms. After you click Show All once the list pops up and even if you click one of the other buttons, the list stays on the form. But also notice that the value of the counters don't change in the list even though you may have clicked Increment a few times and the actual counter values has been upped.

What's happening here is something called Viewstate. Viewstate holds the content of server variables – including labels – in an encoded, hidden form variable that is posted back to the server. So the HTML string is compressed and sent back to the server decoded there and then redisplayed. What this means is that the list that you see if you don't click on the Show All button is a static string rather than a live result retrieved from the COM object. It gets re-posted from its previous form state. As such it gets to be out of sync in this scenario.

There are two ways to solve this issue. You can either check to see if the list is active and if so always reload it from the COM object. Or, probably more efficiently, turn off the Viewstate for the label control and show the list only when the user explicitly asks to see it by clicking on the Show All button which is the approach I took here.

This is a good example of how Viewstate affects your ASP.NET Web forms and data displayed in it – you will find you may have to write a fair amount of code to properly handle data especially if you're dealing with data intensive controls/fields on a form and deciding on whether to support Viewstate on these controls for ease of use or ending up sending a ton of data over the wire to persist the state. A trade off you have to work out for your specific scenario.

Returning non COM objects

One important area is that of returning objects back through COM interop and .NET. Specifically if you want to return objects that aren't defined in the type library of the COM object you are publishing. I do this quite frequently with code shown in Figure 15 (asptools.prg).

Listing 15 (VFP): Returning a record object over COM

```
FUNCTION GetCustObject(lcCustId as String) as Object

IF !USED("TT_Cust")
   USE (THIS.cAppStartPath +"wwdemo\TT_Cust") IN 0
ENDIF

SELE TT_Cust

LOCATE FOR CustNo=PADL(lcCustId,8)
IF FOUND()
   SCATTER NAME loCustomer MEMO
   RETURN loCustomer
ENDIF

SCATTER NAME loCustomer MEMO BLANK

RETURN loCustomer
```

In classic ASP or a COM capable client you can simply do this:

```
Set oVFP = Server.CreateObject("aspdemos.asptools")
Set loCustomer = oVFP.GetCustObject("4")
Response.Write(loCustomer.COMPANY)
```

But this doesn't work in ASP.NET because the SCATTER NAME object is not defined in the Type library that was imported by VS.NET to create the class wrapper. .NET has no idea how to reference the returned object directly.

So how do we do this? The pointer returned is an IDispatch object pointer and we have to access this object indirectly now because .NET can't do the automatic mapping as VBScipt/Jscript ASP was able to do. Unfortunately, this is a bit more work in .NET (Figure 16 – ComInterop.aspx.cs).

Listing 16 (C#): Retrieving a property dynamically

```
private void btnObject_Click(object sender, System.EventArgs e)
{
   ASPToolsClass oVFP = new ASPToolsClass();
   object loCustomer = oVFP.GetCustObject("4");

   // string lcCompany = loCustomer.COMPANY;  // no good!

   object temp =  loCustomer.GetType().InvokeMember(
                  "COMPANY",BindingFlags.GetProperty,null,loCustomer,null);
   string lcCompany = (string) temp;
   this.lblErrorMsg.Text = lcCompany;
}
```

Messy, huh? Unfortunately this is the only way to retrieve values that aren't explicitly available in the type library. As you might expect this mechanism is fairly slow as well, as .NET must look up the type info on the object before retrieving the data. To do so it uses Reflection, which is .NET's extensive type discovery and parsing mechanism. In order to use the above code make sure you add the

```
using System.Reflection;
```

namespace to your source code.

You can simplify the above code a little by using a wrapper method like this (Listing 17 – ComInterop.aspx.cs).

Listing 17 (C#): A simpler way to call a dynamic property with a wrapper

```
private object GetProperty(object loObject,string lcProperty)
{
   return loObject.GetType().InvokeMember(
                lcProperty,BindingFlags.GetProperty,null,loObject,null);
}
```

which reduces the application code to:

```
this.lblErrorMsg.Text = (string) this.GetProperty(loCustomer,"COMPANY");
```

which is considerably more user friendly.

Calling methods dynamically works the same way. The AspTools class includes a method called ReturnObject which can return any object available in the project, whether it's defined as a COM object or not. Let's say you return a full VFP object that isn't a COM object. Listing 18 shows a simulated business object class that simply loads a customer object into an oData member using a Load method (Listing 18 – asptools.prg).

Listing 18 (VFP): Business object that with customer load method

```
DEFINE CLASS aspCustomer as Session

oData = null
cAppStartPath = ""

FUNCTION INIT
*** We need a startup path to find our data
THIS.cAppStartPath = ADDBS(JUSTPATH(Application.ServerName))
SET PATH TO (THIS.cAppStartpath)
DO PATH WITH THIS.cAppStartPath

ENDFUNC

*** Load a customer by ID into the oData member
FUNCTION Load(lcCustId as String) as Boolean

IF !USED("TT_Cust")
   USE (THIS.cAppStartPath + DATAPATH + "TT_Cust") IN 0
ENDIF

SELE TT_Cust

LOCATE FOR CustNo=PADL(lcCustId,8)
IF FOUND()
   SCATTER NAME THIS.oData MEMO
   RETURN .T.
ENDIF

SCATTER NAME THIS.oData MEMO BLANK

RETURN .F.
ENDFUNC
* aspCustomer :: Load

ENDDEFINE
```

We can now use our COM object to return a reference to this class with the ReturnObject method of the ASPTools class:

Listing 19 (VFP): Returning any object from within the project over COM

```
FUNCTION ReturnObject(lcName as String) as Object
oObject = CREATEOBJECT(lcName)
IF VARTYPE(oObject) # "O"
   RETURN .NULL.
ENDIF
RETURN oObject
```

If you now compile the project you can do the following from the VFP command window:

```
oDear = CREATEOBJECT("aspdemos.asptools")
oCust = oDear.ReturnObject("aspCustomer")
? oCust.Load("4")
? oCust.oData.Company
? oCust.oData.Careof
```

We're dynamically creating a VFP object returning it over COM and then call a method on this object, which in turn sets the oData member with the actual values for the retrieved customer.

With .NET we have to use the Reflection interface to perform all these tasks. Let's start by looking at how to make the method call:

Listing 20 (C#): Calling a dynamic method in a returned COM object.

```
ASPToolsClass oVFP = new ASPToolsClass();
object loCustomer = oVFP.ReturnObject("aspCustomer");

string lcCompany =  loCustomer.GetType().InvokeMember("Load",
   BindingFlags.InvokeMethod,null,loCustomer,new object[] {"4"}).ToString();

this.lblErrorMsg.Text = lcCompany;
```

The method call returns a value or true or false which is stored into the label. As with the property retrieval the Reflection functionality is used to dynamically access the COM object and call the method. This time we need to specify a parameter list in an object array which is the last parameter shown here. The single parameter here is a string of "4". As with the property retrieval this code is messy and hard to remember so I created a wrapper method called CallMethod:

Listing 21 (C#): Wrapper method to call a dynamic method on an object
```
private object CallMethod(params object[] loParams)
{
   object[] loActualParams = null;
   object loObject = loParams[0];

   int lnSize = loParams.Length - 2;
   if (lnSize > 0)
   {
      loActualParams = new object[lnSize];

      for (int x=2; x < loParams.Length; x++)
      {
         loActualParams[x-2] = loParams[x];
      }
   }

   return loObject.GetType().InvokeMember(loParams[1].ToString(),
             BindingFlags.InvokeMethod,null,loObject,loActualParams);
}
```

This code relies on C#'s ability to pass variable parameters to a method using a parameter collection that can be parsed at runtime. It takes the object name and method call to be used for the InvokeMember() call and then builds a new object array with all remaining arguments which are the actual parameters to the method call.

Using those two helper methods lets look at how we can handle the Fox code from above in .NET (Listing 22 – ComInterop.aspx.cs).

Listing 22: Accessing a bunsiness object dynamically
```
private void btnObject2_Click(object sender, System.EventArgs e)
{
   ASPToolsClass oVFP = new ASPToolsClass();
   object loCustomer = oVFP.ReturnObject("aspCustomer");

   string lcCompany = "";
   if ( (bool) this.CallMethod(loCustomer,"Load","4") )
   {
      object loData = this.GetProperty(loCustomer,"ODATA");
      lcCompany = (string) this.GetProperty(loData,"COMPANY");
   }

   this.lblErrorMsg.Text = lcCompany;
}
```

Ah, much better. Less code and much more readable and you might actually remember this syntax. As you can see using this sort of logic you can drill down into any kind of object that returns data dynamically but you will have to know its exact interface – there won't be any Intellisense to help you out. This applies not just to FoxPro objects returned as non-COM objects, but also to many system type objects that Windows publishes such as ADSI (Active Directory) and WMI (Windows Management Instrumentation) which publish their provider specific interfaces dynamically without support in type libraries. As long as you know what the names of properties and methods are you can retrieve the data easily.

Note that there a number of other invocation modes availalable including the ability to read 'fields' (as opposed to properties) and set property/field values both on .NET and COM objects dynamically. All this is possible through the Reflection interface that .NET publishes and this provides powerful runtime support for type discovery.

Passing ASP.NET objects to VFP COM objects

If you've used COM with classic ASP you might recall that ASP used to publish a couple of interfaces that made it possible to access the ASP intrinsic objects such as Request, Response, Server, Application and Session etc. directly from within VFP code. With ASP.NET this functionality is still supported through the ASPCOMPAT directive on an ASP page.

The IScriptingContext interface no longer works. IScriptingContext in classic ASP caused two methods – OnStartPage and OnEndPage – to fire when an ASP request occurred. OnStartPage automatically passed Context object as a parameter. The Context object provides the ability to retrieve the various intrinsic objects. This interface no longer appears to work, even though the ASP.NET docs state that it should work (no word on whether this is a bug in .NET or VFP).

More recently Microsoft recommended using the ObjectContext interface instead of IScriptingContext, and this interface properly works with VFP COM objects. The ObjectContext class provides the same functionality as the IScriptingContext Context container object in addition to provide COM+ services and transaction management. The ObjectContext object is based and uses the COM+ services to provide the functionality to VFP.

To access the object in VFP and retrieve ASP object you can use the following method of the AspTools class example as a reference (Listing 23 – asptools.prg).

Listing 23 (VFP): Accessing ASP.NET/ASP intrinsic object from VFP COM code

```
FUNCTION ASPObjects() as Void

*** Grab the object context
oMTS  = CreateObject("MTxAS.AppServer.1")
oContext = oMTS.GetObjectContext()

*** Retrieve some of the ASP objects
loRequest = oContext.item("Request")
loResponse = oContext.item("Response")
loSession = oContext.item("Session")

lcOutput=""
THIS.cErrorMsg=""

*** Query String
lcOutput = "<PRE>QueryString (Method): " + loRequest.QueryString("Method").item()+"<br>"

*** Full QueryString
lcOutput = lcOutput + "Full QueryString: " + loRequest.QueryString("").item() + "<BR>"

*** Server Variable
lcOutput = lcOutput + "Browser: " + loRequest.ServerVariables("HTTP_USER_AGENT").item()
+ "<BR>"

*** Form Variable
lcOutput = lcOutput + "Form Company: " + loRequest.Form("txtCompany").item() + "<BR>"

*** Accessing and assigning Session Vars
lcOutput = lcOutput + "Session VFPCreatedSession: "
IF !ISNULL(loSession.Value("VFPCreatedSession") )
   *** Show it if it exists
   lcOutput = lcOutput +  loSession.Value("VFPCreatedSession") + "<p>"
ELSE
   *** Otherwise set the session var
   loSession.Value("VFPCreatedSession") = "VFPCreatedSession Value " + TIME()
ENDIF

lcOutput = lcOutput + "<h2>Query String</h2>"
loQueryString = loRequest.QueryString
FOR EACH lcFormVar in loQueryString
   lcOutput = lcOutput + lcFormVar + "=" + ;
              loRequest.QueryString(lcFormVar).Item() + CR
ENDFOR

lcOutput = lcOutput + "<h2>Server variables</h2><PRE>" + CR
loServerVars = loRequest.ServerVariables
FOR EACH lcFormVar in loServerVars
   IF lcFormVar="ALL_HTTP"
      LOOP
   ENDIF
   lcOutput = lcOutput + lcFormVar + "=" + ;
              loRequest.ServerVariables(lcFormVar).Item() + CR
ENDFOR

*** Form Vars
lcOutput = lcOutput + "<h2>Form Variables</h2>" + CR
loFormVars = loRequest.Form
FOR EACH lcFormVar in loFormVars
   lcOutput = lcOutput + lcFormVar + "=" + ;
              loRequest.Form(lcFormVar).Item() + CR
ENDFOR

lcOutput = lcOutput + "<PRE>"

loResponse.Write(lcOutput + THIS.cErrorMSg)
RETURN
```

To call this from ASP.NET simply use the code in Listing 23.1 (AspObjects.aspx).

Listing 23.1 (C# ASP.NET): Sample form to demonstrate ASP objects in VFP

```
<%@ Page ASPCOMPAT=true language="c#" Codebehind="AspObjects.aspx.cs"
AutoEventWireup="false" Inherits="ASPInterOp.AspObjects" %>
<html>
    <h2>VFP access to ASP intinsic objects</h2>
    <hr>
    <form action="AspObjects.aspx?Method=Test&Parm2=Another+Parm" METHOD="POST">
          LastName: <input name="txtLastName" value='<%=
Request.Form["txtLastName"]%>'><br>
          Company: <input  name="txtCompany" value='<%=
Request.Form["txtCompany"]%>'><p>
                <input type="Submit" name="btnSubmit">
    </form>

<%
if (Request.Form["btnSubmit"] != null )
{
  Response.Write("<hr><i>Generated from within Visual FoxPro:</i><p>");
  ASPToolsClass oVFP = new ASPToolsClass();
  oVFP.ASPObjects();
}
%>

</html>
```

The code in the page checks to see to only access the Fox server if the form was submitted, then calls the method and echos back the request information in HTML format as generated by the Fox server.

For those of you familiar with classic ASP and COM note that there are a few subtle differences in behavior of the various objects provided. For example, you can no longer access the various collections without parameters. For example, Request.QueryString() used to return the full querystring in ASP, but fails in ASP.NET. Basically any access of a collection item that doesn't exist returns a NULL rather than a null string (""), which means you may need additional error handling if you can't guarantee that a value is actually available. Otherwise things like this:

```
Request.Form("txtUnPostedVar").Item()
```

Will fail with a data type error as the Item method doesn't exist on a null reference.

Furthermore be aware that if you use the Response object in combination with ASP.NET Web forms that the results may not be exactly what you expect. Web Forms do not use the standard Response stream and place text only after the entire page has rendered. So if you use Response.Write before the page has rendered the output from it will likely end up at the very top of the page before the <html> tag of the ASP.NET page. Using Response.Write from within VFP thus only makes sense if the COM object will generate the entire HTTP output, or if the calling ASP.NET page also uses Response output rather than the Web Forms engine (ie. no Server Controls).

Passing DataSets as XML between VFP and .NET

As described earlier.NET uses a new data access metaphor in the ADO.NET architecture by utilizing DataSets and XML to pass data around applications. For VFP applications this means that the easiest way to pass data to .NET often is in the form of XML. VFP's CURSORTOXML() is capable of exporting data in a format that .NET understands and can import into a DataSet fairly easily. Consider the following method returning a cursor in XML format to the ASP.NET page(Listing 23.1 – asptools.prg).

Listing 23.1: Returning XML from a VFP query

```
FUNCTION XMLCustList(lcCompany as String) as String

*** VFP Sample Data
lcDataPath = VFPSAMPLEDATA
this.cDataPath = lcDataPath

lcWhere = ""
IF !EMPTY(lcCompany)
   lcWhere = " WHERE Company = lcCompany "
ENDIF

SELECT company,contact,PADR(TRIM(city) + ", " + Country,60) as Location,Phone ;
   FROM (lcDataPath + "Customer") ;
   &lcWhere ;
   ORDER BY company ;
   INTO CURSOR TQuery

lcXML = ""

CURSORTOXML(ALIAS(),"lcXML",1,32,0,"1")

RETURN lcXML
```

To use this data in .NET in a DataSet you can load a new DataSet object from the returned XML using code like the following (Listing 23.2 – ComInterop.aspx.cs).

Listing 23.2: Loading a DataSet from XML returned by VFP

```
private void btnDataSet_Click(object sender, System.EventArgs e)
{
   ASPToolsClass oVFP = new ASPToolsClass();
   string lcXML = oVFP.XMLCustList("");

   DataSet loDS = new DataSet();
   loDS.ReadXml( new StringReader(lcXML) );

   this.oGrid.DataSource = loDS.Tables[0];
   this.oGrid.DataBind();
}
```

The ReadXml method of the DataSet can read XML returned from VFP and import into a DataTable with all of the type information intact. For example, if you returned the MaxOrdAmt field you can check the type with:

```
this.lblErrorMsg.Text = loDS.Tables[0].Rows[0]["MaxOrdAmt"].GetType().Name;
```

which returns 'decimal' which is .NET's translation for the currency type in VFP.

What about passing data the other way around? From .NET to VFP? I'll show you how to pass a DataSet from .NET to VFP and manipulate the data directly out of the DataSet a little later in the 'Calling .NET components from VFP section'. This process works, but can be tricky to work with because you have to know the exact interfaces of the DataSet and the exact layout of the data.

You can also pass XML as a string back to the VFP application. Unfortunately VFP has no built in way to utilize this XML directly using say XMLTOCURSOR(). There are a couple of problems with turning DataSets into VFP data – the schema information of the datasets do not necessarily contain all the data needed to create a VFP cursor from the data, so a parser would have to make two passes through the data to create the field sizes. This can create problems with field sizing and has potential performance problems.

One way that you can import DataSets in VFP from XML is to use wwXML with some manual intervention. Consider the following ASP.NET page that creates a DataSet from the SQL Server Pubs database then passes the DataSet to VFP via an XML string (Listing 23.3 – ComInterop.aspx.cs).

Listing 23.3 (C# ASP.NET): Passing DataSet XML to a VFP COM method

```
private void btnDataSetToXML_Click(object sender, System.EventArgs e)
{
   string lcID = "%";

   DataSet ds = new DataSet();

   string cConnection = "server=(local);database=pubs;uid=sa;pwd=";
   SqlConnection   oConn = new SqlConnection(cConnection);

   SqlDataAdapter oAdapter = new SqlDataAdapter();
   oAdapter.SelectCommand =
      new SqlCommand("select * from Authors where au_id like '" +
                     lcID + "%'",oConn);

   oConn.Open();

   oAdapter.Fill(ds,"authors");

   oConn.Close();


   // *** This code passes the DataSet to VFP as XML
   ASPToolsClass oVFP = new ASPToolsClass();

   StringWriter loWriter = new StringWriter();
   ds.WriteXml( loWriter );

   // *** VFP returns an HTML table of the data passed to the method
   this.lblCounterList.Text = oVFP.XMLDataSet(loWriter.ToString(),"authors");
}
```

The code in the VFP COM object uses the XML to create a cursor, and then just to demonstrate that it got the data returns it as an HTML table. You can use wwXML (included with the samples) which has a method called DataSetXMLToCursor to create a VFP cursor from an individual DataTable contained within a DataSet. Because of the limitations of the schema published by a DataSet, wwXML requires that you pre-create the cursor that the data is imported to (Listing 23.4 – asptools.prg).

Listing 23.4 (VFP): Importing a DataSet from XML using the wwXML class.

```
FUNCTION XMLDataSet(lcXML as String,lcCursor as String) as String

oXML = CREATEOBJECT("wwXML")
CREATE CURSOR Temp (au_id c(10), au_lname c(15), Contract L, PK I)
oXML.DataSetXMLToCursor(lcXML,"temp","authors")

*** Just echo the data back as an HTML string
RETURN this.ShowCursor()
```

Note the CREATE CURSOR command to pre-create the table that you're importing to. wwXML uses the structure of the table to import the XML data into the cursor thus avoiding some of the schema limitations. But this limitation also means that you have to know what type of data you're expecting to import beforehand.

Debugging your COM server

Debugging in .Net is just as complicated as it is in ASP classic as the COM object is a compiled application that runs inside of IIS, so by default you can't debug the COM server. However, with a little trickery you can actually debug your components by using the VFP IDE to instantiate the object for you and return it to the ASP.Net page. This idea was originally from Maurice de Beijer who posted an interesting article on how to pass control to VFP via Accessibility objects (http://www.theproblemsolver.nl/aspdevelopmentanddebugging/index.htm).

I had problems with this approach and found a slightly different, but similar way to accomplish the same thing by adding a property and a couple of methods to my COM server:

Listing 23.5 (VFP): Returning a debuggable instance of a VFP COM Object

```
oVFPDebug = .NULL.
…
************************************************************************
* ASPTools :: GetDebugInstance
****************************************
***  Function: Allows debugging of the server
***            Pops up debugger on SET STEP or dialog on error
************************************************************************
FUNCTION GetDebugInstance as ASPTools
LOCAL loObject

THIS.oVFPDebug = CREATEOBJECT("VisualFoxPro.Application.7")
THIS.oVFPDebug.Visible = .T.
THIS.oVFPDebug.DoCmd("cd " + THIS.cAppStartPath)

*** Loader for classlibs (header of this file)
*** Makes sure that the class is found and can load
THIS.oVFPDebug.DoCmd("DO ASPTools")

loObject = THIS.oVFPDebug.Eval("CREATE('asptools')")
RETURN loObject
ENDFUNC
*  ASPTools :: GetDebugInstance

************************************************************************
* ASPTools :: CloseDebugInstance
****************************************
***  Function: Shuts down the debug instance created with
***            GetDebugInstance
************************************************************************
FUNCTION CloseDebugInstance()

THIS.oVFPDebug.visible = .F.
THIS.oVFPDebug = 0

ENDFUNC
*  ASPTools :: CloseDebugInstance
```

This approach basically instantiates an instance of the Visual FoxPro IDE as a COM object and creates an instance of the object and returns that instance back to the ASP.Net page.

To access this functionality from .Net you'd use code like this:

Listing 23.6 (C#): Using a debuggable instance of the VFP COM component

```
ASPToolsClass oVFP = new ASPToolsClass();
object loVFP = oVFP.GetDebugInstance();

// *** casting doesn't work as we have an IDispatch pointer
// ASPToolsClass loTemp = (ASPToolsClass) loTemp;

// *** We'll have to use indirect calls with Reflection
string lcHtml = (string) this.CallMethod(loVFP,"HelloWorld");
int lnId =(int) this.CallMethod(loVFP,"GetProcessId");

oVFP.CloseDebugInstance();
```

Voila – you can debug your VFP COM server in real time. Set a breakpoint in any of the methods called and the debugger will pop up. Note that you will not be able to directly access the new reference and have to use the indirect method calling mechanisms described earlier like CallMethod() and GetProperty() if you are using C# or VB.Net with Option Strict. The reason for this is that the ASPToolsClass is a wrapper for the actual COM object, but not the COM object itself. This means the COM object can't be cast to an ASPToolsClass object and you need to use indirect method and property access.

However, with this in place you can now set breakpoints in your code and the debugger will pop up for you as needed. Any failures too will pop up an error dialog in the VFP IDE and you can examine the state of the application in the debugger. While not optimal because of the indirect object reference requirement for you object, this should nevertheless be a big help in debugging COM components. FWIW, this process would be easier in Visual Basic which can deal with untyped COM objects when Option Strict is off.

This works in classic ASP as well, and is in fact a little easier.

```
<%
SET oVFP = Server.CreateObject("aspdemos.asptools")

set loVFP = oVFP.GetDebugInstance
Response.Write( loVFP.IncCounter("EssentialFox",0) )

oVFP.CloseDebugInstance()
%>
```

Here you can call methods directly and you can simply rename your main server reference to something else and assign the debug object to the object var instead which means you can debug by simply adding the GetDebugInstance() and CloseDebugInstance() calls.

ASP.NET and VFP COM object performance

ASP.NET changes the mechanism of how COM objects are accessed considerably compared to classic ASP. As a result COM interop from ASP.NET tends to be somewhat slower than it was with classic ASP.

The main difference is that ASP.NET uses the Multi-Threaded Apartment (MTA) model to run internal components, while classic ASP uses the Single Threaded Apartment (STA) model. Visual FoxPro (and Visual Basic) COM objects are STA components – each object instance runs on a single thread and must always be accessed on the same thread that created it. MTA components can run on any thread and be invoked from different threads, because they are assumed to be thread safe. The MTA provides for potentially higher performance as the system has to do much less work to fire a method of an object as it doesn't have to keep track of which thread created the component. The MTA model is a big reason for the improved performance of ASP.NET as a Web backend engine in combination with the compiled nature of the code that it runs.

Natively ASP.NET runs in the MTA model and in order to run VFP/VB components the ASPCOMPAT page directive is required to switch the page in STA model operation. This has several implications. First pages that use ASPCOMPAT must be managed differently than pages that are running in MTA – the system once again must track threads and make sure components run on the proper threads that they were created on. This switch affects performance of the individual page as well as the overall ASP system while any pages that require ASPCOMPAT are active.

In addition, ASP.NET is optimized for managed code and calling out into unmanaged COM components requires some overhead.

All of this is to say that running COM components in ASP.NET is going to be slower than running them in classic ASP applications. I ran a few tests on the IncCounter example I previously mentioned. To keep things simple I only used Response.Write() output code when I tested the performance with roughly identical ASP and ASPX pages. I tried the component both with and without registering it under COM+. To test I ran Visual Studio Test Center, created a test script with just a sinlge page in it (the ASP or ASPX page). I then ran through 5 sets of 5 minute tests against a single URL and took the average value from these tests (which were very consistent BTW). The following Test Machine was used: Pentium III 1ghz, 512 megs memory Windows 2000 Advanced Server SP2). I also made sure that the ASP.NET application was re-added to the .NET project and recompiled each time the component was moved in and out of COM+. The ASP .NET application also was set to Release mode and the configuration settings with debug settings turned off – IOW, optimized for production testing (incidentally not doing so as I did when I ran the first few tests didn't affect performance all that much – less than 2% on these test).

I kept the tests really simple to measure mainly the COM overhead. The classic ASP code is just:

```
<%
SET oVFP = Server.CreateObject("aspdemos.asptools")
Response.Write( oVFP.IncCounter("EssentialFox",0) )
%>
```

The ASP.NET code is:

```
private void Page_Load(object sender, System.EventArgs e)
{
   ASPToolsClass oVFP = new ASPToolsClass();
   Response.Write( oVFP.IncCounter("EssentialFox",0) );
}
```

Here are the results:

	Classic ASP	**ASP.NET**
Plain COM	118 rps	99 rps
COM+ Component	61 rps	67 rps

Notice that classic ASP is about 20% faster for plain operation. It's interesting that the COM+ component access is considerably slower both for classic ASP and ASP.NET so much so that the overhead probably removes the actual performance differences between Classic and ASP.NET. In fact with COM+ components .NET is actually slightly faster.

The ASP code here is very simple – it simply instantiates the object and does a Response.Write() on the result from IncCounter(). This demonstrates basically the call overhead of the methods plus a basic data access operation in the VFP code. The numbers change drastically if you now add a few calls to these methods

```
<%
SET oVFP = Server.CreateObject("aspdemos.asptools")
Response.Write( oVFP.IncCounter("EssentialFox",0) )
Response.Write( oVFP.IncCounter("EssentialFox1",0) )
Response.Write( oVFP.IncCounter("EssentialFox2",0) )
Response.Write( oVFP.IncCounter("EssentialFox3",0) )
Response.Write( oVFP.IncCounter("EssentialFox4",0) )
%>
```

	Classic ASP	ASP.NET
Plain COM	118 rps	84 rps
COM+ Component	48 rps	58 rps

Notice how the performance of classic ASP stays almost stable even with the additional COM calls- it appears that most of the overhead is in the creation of the object with very little overhead for additional method calls. Calling methods is fairly fast and doesn't affect performance much. With ASP.NET however, repeated COM calls cause additional overhead as code is crossing the managed to unmanaged boundaries on each COM method call or property access performance drops significantly for every added COM call.

But look at the COM+ numbers. In all scenarios running a COM+ component seems to slow down performance significantly and the difference between ASP and ASP.NET actually switched to an advantage for .NET. But performance stays very steady regardless of accesses to the component and in fact ASP.NET is faster with COM+ components.

Regardless, of the slower numbers with .NET keep in mind that the even the lower numbers are rather acceptable in terms of performance! If you can sustain even the slowest 40 requests a second on a single server is still close to 3.5 million hits in 24 hour period, which is impressive for the test hardware it's running (midrange desktop machine w/o fully optimized hardware).

But regardless it's important to understand that there is a performance penalty to pay when using ASP.NET over classic ASP. In fact, if you're building an application that's primarily using COM objects to perform their business logic or even have applications that serve all your HTML content from within a COM object (some backend engines use ASP as a Web front end) classic ASP may actually be a much better choice for performance reasons. Comparing plain COM operation with lots of method calls to COM components you can see that performance can multiple times faster with classic ASP.

Keep in mind that this test mainly tests for COM call overhead not the actual speed of the code running. As you run requests that run more code inside the VFP component the call overhead becomes less important. But if you're accessing lots of properties or short quick method calls (which often is typical for business object access) you will see performance issues similar to this test.

Accessing .NET objects through COM

In the last section I showed you how you can integrate COM objects into .NET – now let's look at going the other way by creating objects or accessing native .NET objects from within a COM client application like Visual FoxPro.

Do you need .NET from your VFP apps?

Before you do this, you should think hard about your reasons for doing so. .NET provides many new features in an easy to access mechanism, but realize that you can probably access that same functionality through other non-.NET mechanisms as well. If your only goal is to access some feature that .NET provides you're probably better of skipping it. The other scenario of course is true interop between two applications or operation between an application upgrade that is gradually being moved to .NET.

If you integrate .NET into your VFP application you should realize that you will incur the full requirements of the .NET platform:

- **.NET Framework must be installed**
  You have to make sure that the framework is installed and if your application is distributed it will have to probably provide the 25meg or so runtime with it as few machines to date have the framework installed. You have to make sure that the proper version of the .NET runtime is installed if you don't install it yourself.
- **Hardware requirements**
  The .NET framework requires fairly heft hardware to run on especially memory. Typical .NET components will incur about 30 meg memory footprint for the first one loaded (which is the runtime, plus .NET services, plus your actual components footprint. Additional components or EXEs will have smaller footprints.
- **Registration of components**
  .NET COM components don't register like standard COM components and must use .NET specific tools (RegAsm.exe or the appropriate runtime libraries) to register a component.

The point is that by including .NET with your VFP application, you're essentially requiring two separate development runtimes to be installed and in some cases it may actually be better to go all the way to .NET or stick completely with VFP/Win32/COM.

Publishing a .Net component to COM

As with COM interop from .Net using .Net objects from VFP appears to be a simple matter of creating a class and publishing the class as a COM object. However, there's more to it than this especially if you are planning to pass native .Net objects back to VFP. The type differences between COM and .Net make it difficult and sometimes not possible to access all the features that the .Net framework offers in your VFP applications.

This means that in many cases you may not be able to directly publish your .Net business or system classes into COM, but rather require wrappers that expose the functionality in a COM compatible way. I'll give an example of this concept a little later.

But let's start with a cool example that I had use for on my Web site: A small component to create thumbnail images from existing image files as shown in Listing 24.

Listing 24 (C#): A class method to create a thumbnail image from an image

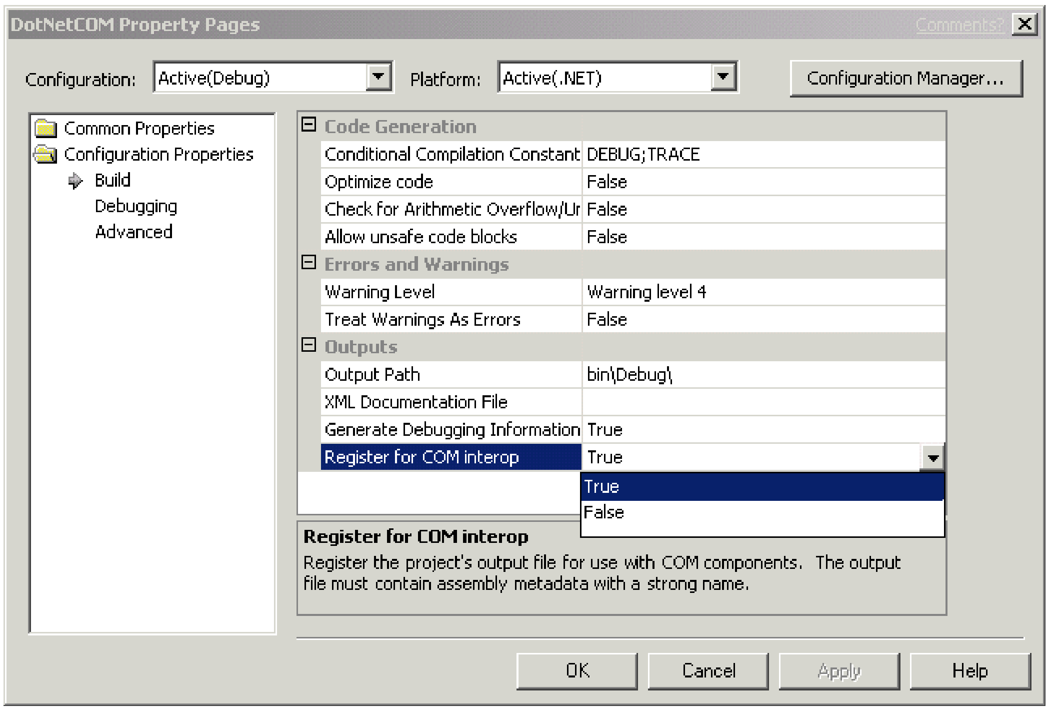
```
using System;
using System.Drawing;
using System.Drawing.Imaging;

using System.Runtime.InteropServices;

namespace DotNetCOM
{
   [ClassInterface(ClassInterfaceType.AutoDual)]
   [ProgId("DotNetCOM.wwImaging")]
   public class wwImaging
   {
      public bool CreateThumbnail(string lcFilename, string lcThumbnailFilename,
                                  int lnWidth, int lnHeight)
      {
         try
         {
            Bitmap loBMP = new Bitmap(lcFilename);
            ImageFormat loFormat = loBMP.RawFormat;

            decimal lnRatio;
            int lnNewWidth = 0;
            int lnNewHeight = 0;

            //*** If the image is smaller than a thumbnail just return it
            if (loBMP.Width < lnWidth && loBMP.Height < lnHeight)
            {
               loBMP.Save(lcThumbnailFilename);
               return true;
            }

            if (loBMP.Width > loBMP.Height)
            {
               lnRatio = (decimal) lnWidth / loBMP.Width;
               lnNewWidth = lnWidth;
               decimal lnTemp = loBMP.Height * lnRatio;
               lnNewHeight = (int)lnTemp;
            }
            else
            {
               lnRatio = (decimal) lnHeight / loBMP.Height;
               lnNewHeight = lnHeight;
               decimal lnTemp = loBMP.Width * lnRatio;
               lnNewWidth = (int) lnTemp;
            }

            System.Drawing.Image imgOut =
                      loBMP.GetThumbnailImage(lnNewWidth,lnNewHeight,
                                              null,IntPtr.Zero);

            loBMP.Dispose();

            imgOut.Save(lcThumbnailFilename,loFormat);
            imgOut.Dispose();
         }
         catch (Exception ex)
         {
            this.SetError(ex.Message);
            return false;
         }

         return true;
      }
      protected void SetError(string lcError)
      {
         if (lcError.Length == 0)
         {
            this.cErrorMsg = "";
            this.bError = false;
         }
         else
         {
            this.cErrorMsg = lcError;
            this.bError = true;
         }
      }
   }

}
```

CreateThumbnail() takes input and output filenames and the size for the thumbnail to be created. Rather than sizing the image absolutely and distorting the image this method tries to maintain the aspect ratio by figuring out which side to reduce the passed parameter. So if the picture is wider the width value is used if it is higher the Height value is used. The bulk of the code deals with calculating this ratio and the right size for the image to create. The actual image conversion is actually handled natively through the .Net framework by using the CreateThumbnailImage method of the bitmap or image objects which creates a new image object that can then simply be saved to disk with the Save method. CreateThumbnailImage takes the Height and Width parameters and a pointer to a callback function (null) and another pointer to a data block which is uncharacteristically typed as a Void * (an unmanaged type). The latter needs to be passed as IntPtr.Zero rather than null.

I'm using a couple attributes on the class to explicitly force several options of how the COM class interface is published. If these two options were left out defaults would be used. The ClassInterfaceType.AutoDual forces the COM object to be created using dual interfaces which creates both IDispatch (Late binding) and CoClass (Early binding) interfaces. The default is AutoDispatch, which works fine for VFP usage, but doesn't provide the interfaces necessary to use Intellisense, so overriding this attribute is important. The alternative is to explicitly create an Interface for all published methods of the class, then create a separate class with the implementation, which is obviously more tedious. Unless you have a specific need for the interface definition there is no need to go this route.

The ProgId is also overridden explicitly – the default is the namespace plus the name of the class, which in the case above is exactly the same as the default. I like to override this because in many cases the namespace may not be the name I would choose for the server's ProgId.

The attributes of the class are essential to have types exported into COM. In addition you need to force the .Net framework to actually emit the type library. When this code compiles it really only creates a standard .Net class. If you're using Visual Studio.Net you can simply specify that you want your project to be compiled as a COM class (Figure 5).



Configuration: Active(Debug)   Platform: Active(.NET)

Code Generation	
Conditional Compilation Constant	DEBUG;TRACE
Optimize code	False
Check for Arithmetic Overflow/Ur	False
Allow unsafe code blocks	False
Errors and Warnings	
Warning Level	Warning level 4
Treat Warnings As Errors	False
Outputs	
Output Path	bin\Debug\
XML Documentation File	
Generate Debugging Information	True
Register for COM interop	True

True
False

Register for COM interop
Register the project's output file for use with COM components. The output file must contain assembly metadata with a strong name.

Figure 5 *– Visual Studio.Net can automatically compile your project into a COM component by setting the 'Register for COM interop' option in the Project's Build options. To get this dialog right click on the project and select Properties.*

If you don't use VS.Net you can the *RegAsm.exe* utility to perform this same functionality. Both mechanisms create a type library for all the types defined in your project and create the appropriate registry entries.

The class attributes in the source code determine how the additional proxies used by the .Net COM runtime are invoke the object. Unlike other COM objects, if you check the ClassId in the registry you'll find it doesn't actually point at your DLL, but rather at a generic .Net framework DLL – mscoree.dll. Other keys then define the location of the actual DLL that the generic .Net COM object runtime must load and invoke. Figure 6 shows what the registry entry looks like.

Figure 6 *– The registry entry for the .Net COM component points at a generic runtime COM dll – mscoree.dll, rather than your DLL. The COM runtime loads the actual assembly and sets up the appropriate proxies.*

Returning and accessing Objects from .Net

You can also return complex data like objects from .Net. Let's add another method to the class called GetImageInfo that returns the Height and Width and Screen Resolution of an image by returning an object as shown in Listing 25:

Listing 25 (C#) – Returning info about an image as an object

```
public wwImageInfo GetImageInfo(string lcImageFile)
{
   try
   {
      Bitmap loBMP = new Bitmap(lcImageFile);

      wwImageInfo loInfo = new wwImageInfo();

      loInfo.Width = loBMP.Width;
      loInfo.Height = loBMP.Height;
      loInfo.HorizontalPixelResolution = loBMP.HorizontalResolution;
      loInfo.VerticalPixelResolution = loBMP.VerticalResolution;

      loBMP.Dispose();

      return loInfo;
   }
   catch(Exception ex)
   {
      this.SetError(ex.Message);
   }

   return null;
}
```

The object is defined as follows:

```
public class wwImageInfo
{
   public int Height = 0;
   public int Width = 0;
   public float HorizontalPixelResolution = 0;
   public float VerticalPixelResolution = 0;
}
```

Now recompile the class in VS.Net. The first thing that you'll notice is that you won't be able to do this if VFP is still running. Even if you released the object in Visual FoxPro the object will remain locked in memory.

According to Microsoft the COM Callable Wrapper that instantiates COM objects will never release the objects that have been loaded by unless the Interop layer explicitly forces these objects to unload. This needs to be done on a low COM level using a specific interface that .Net components can publish IDisposable. However, this cannot be directly accessed through VFP so there's no way to unload .Net COM objects at this time. No way except shutting down Visual FoxPro that is. For what it's worth the same is true of Visual Basic and other COM clients. This may be addressed in future versions but it is an issue now. Note that if you load a .Net COM object you incur the memory hit of the .Net runtime (about 3 megs) once. Any subsequent .Net objects loaded will have very little memory impact.

Ok, so shut down VFP and start her back up. Then try this code:

```
o = CREATEOBJECT("DotNetCom.wwImaging")
```

```
loImage = o.GetImageInfo("d:\westwind\wconnect\wcpower.gif")
? loImage.Height
? loImage.Width
? loImage.HorizontalPixelResolution
```

You've just retrieved the content of the object passed back from .Net. But notice that you didn't get Intellisense on the returned object. The reason for this is that we need to provide the proper attributes just like for the main wwImaging class in order to bring the type information interfaces back for VFP to evaluate. Change the class header to:

```
[ClassInterface(ClassInterfaceType.AutoDual)]
class wwImageInfo {... rest of class here}
```

Then shut down VFP and recompile the class and try the above code again. Voila, Intellisense now works.

Returning .Net built-in objects

What about .Net objects? If you look at the code for the GetImageInfo method you might ask yourself, why this separate object? Why not just pass back the full Bitmap object and let VFP deal with it directly?

To see what happens if you add the following method to the class that returns the bitmap directly (Listing 26 – wwImaging.cs).

Listing 26 – Returning a .Net Bitmap object

```
public Bitmap GetImageInfoBitmap(string lcImageFile)
   {
      try
      {
         Bitmap loBMP = new Bitmap(lcImageFile);

         wwImageInfo loInfo = new wwImageInfo();

         loInfo.Width = loBMP.Width;
         loInfo.Height = loBMP.Height;
         loInfo.HorizontalPixelResolution = loBMP.HorizontalResolution;
         loInfo.VerticalPixelResolution = loBMP.VerticalResolution;

         return loBMP;
      }
      catch(Exception ex)
      {
         this.SetError(ex.Message);
      }

      return null;
   }
```

Then try the following code with a reasonably large image in VFP:

```
o = CREATEOBJECT("DotNetCom.wwImaging")
FOR x = 1 TO 100
   loBMP = o.GetImageInfoBitmap(
             "\westwind\rick\photoalbum\maui_2002\campingLaPerouse_big.jpg")
   lnHeight =loBMP.Height
ENDFOR
RETURN
```

Before you run this actually bring up TaskManager and highlight VFP in it. As you run keep an eye on the memory usage! In my test with a 40k JPG file running this code resulted in VFP ending up with over 200 megs of memory used!!!

This means that even though we are releasing the object in VFP, it's not completely releasing in .Net. Keep Task Manager running though, and run maybe the CreateThumbnail() method again a few times. Notice that at some point the memory does get released. This is .Net's garbage collector eventually freeing the memory for the objects in use – so the objects release eventually, just not right away.

To avoid the enormous resource use in this example above you have to manually release resources used by the bitmap object by doing:

```
loBMP.Dispose()
```

Like most .Net classes Bitmap supports a Dispose method that cleans up resources used, although it doesn't mean that the object reference is released or 'disposed' at all. Dispose is .Net talk for 'Object: clean up behind yourself'. Adding loBMP.Dispose() before the ENDFOR keeps memory usage reasonable by cleaning up the bitmap data even if the object reference itself is not released inside the .Net runtime that manages the lifetime of the COM called .Net object.

Because of the release issues you'll want to consider carefully if you want to pass .Net objects to VFP especially in looping situation as this will create large numbers of object references in the .Net AppDomain that the COM Interop layer creates. These objects might not immediately release until the garbage collector gets to them. Be aware of what resources are in use and if the object is not passed back to .Net it's a good idea to call Dispose() (if available) or any other cleanup methods that the object might provide.

Passing objects to .Net methods

Assume for a minute that you want to pass an object from Visual FoxPro to your .Net application like this:

```
USE tt_cust
SCATTER NAME loCustomer MEMO

o = CREATEOBJECT("DotNetCom.DotNetComPublisher")
? o.ReturnCustomer(loCustomer)   && prints content of Company field
```

If you'll remember from the previous section on retrieving dynamic properties in .Net, passing objects from VFP to .Net results in a generic object that isn't defined in the type library. And as before we have to resort to retrieving the content and accessing methods of the passed object in the .Net code using the Reflection InvokeMember interfaces.

Here's how to implement this method in .Net using the GetProperty method mentioned earlier to simplify access to properties:

```
public string ReturnCustomer(object loObject)
{
    //*** loObject is passed from VFP – retrieve Company field and return
    return (string) this.GetProperty(loObject,"Company");
}
```

You'll have to use the same type of thing for methods using the CallMethod() method introduced earlier.

Is this always necessary? It depends on whether you provide an import COM type library in the .Net project or not. If you want to be able to pass a known type that is published in the VFP code as a COM object then you can import the type library and reference that type explicitly. But if you simply pass a 'generic' object you will always have to use Reflection's InvokeMember (or the GetProperty/GetMethod shortcuts) to access the object's members.

Publishing types via COM is a lot of work though especially if your application isn't already a COM object of some sort – for example a standalone desktop application likely will not publish anything to COM. But you might consider the extra effort of exposing objects via COM even if you otherwise wouldn't to allow the type library to be published and have it accessible to .Net. In addition to easier type access you also gain the ability to access the type in .Net with Intellisense which is otherwise lost to you.

Working with the DataSet object

So far it looks very easy to export objects from .Net to VFP, right? And it is straight forward with your own objects and code. However, things get more tricky when you start dealing with the some of the complex objects that .Net itself publishes.

For example consider the following method that returns an ADO.Net DataSet from the SQL Server Pubs database (Listing 27 – DotNetComPubliser.cs).

Listing 27 (C#): Method that returns a DataSet

```
// …add namespace: using System.Data.SqlClient

public System.Data.DataSet GetAuthorData(string lcID)
{

   if (lcID == "" )
      lcID = "%";

   DataSet ds = new DataSet();

   string cConnection = "server=(local);database=pubs;uid=sa;pwd=";
   SqlConnection   oConn = new SqlConnection(cConnection);

   SqlDataAdapter oAdapter = new SqlDataAdapter();
   oAdapter.SelectCommand =
      new SqlCommand("select * from Authors where au_id like '" + lcID +
                     "%'",oConn);

   try
   {
      oConn.Open();
   }
   catch(Exception e)
   {
      return null;
   }

   oAdapter.Fill(ds,"Authors");


   oConn.Close();
   return ds;
}
```

Recompile the DotNetCom project again and now go back into Visual FoxPro and try the following code:

```
o = CREATEOBJECT("DotNetCom.DotNetComPublisher")
loDs = o.GetAuthorData("172-32-1176")
? loDS.GetXml()
```

This code retrieves the DataSet from the Pubs database and then retrieves an XML string of the data. We've retrieved a DataSet object from .Net and are manipulating it in Visual FoxPro. You can access the methods and properties of the Dataset as you would expect. However, also notice that Intellisense is not working on the Dataset because .Net wasn't explicitly asked to export it. This makes figuring out what properties are available of the full Dataset object a little tricky.

In addition, collection based values are not accessible in the same way as you can from inside of .Net. The following **does not** work:

```
? loDS.Tables["Authors"].Rows[0]["au_lname"]  && Doesn't work!!!
```

Named collections are common to native .Net objects, but I've been unable to use named collection items through VFP. You can however use them through positional collection values. The following does work:

```
? loDS.Tables.Item(0).Rows.Item(0).Item(1)
   && au_lname in the first row
```

There may be another way to get at the collection with names but I couldn't find it. This at least allows access to the features of the DataSet even though it can be hard to maintain code like this.

To demonstrate how to use a DataSet object in a VFP application I created a small sample application against the PUBS SQL Server sample database that retrieves a DataSet and allows displaying and editing of the content inside of VFP. The updates can then be sent back to the .Net COM object to save the changes into the SQL Server database. This sample can be found in the ComAuthors.scx form file.

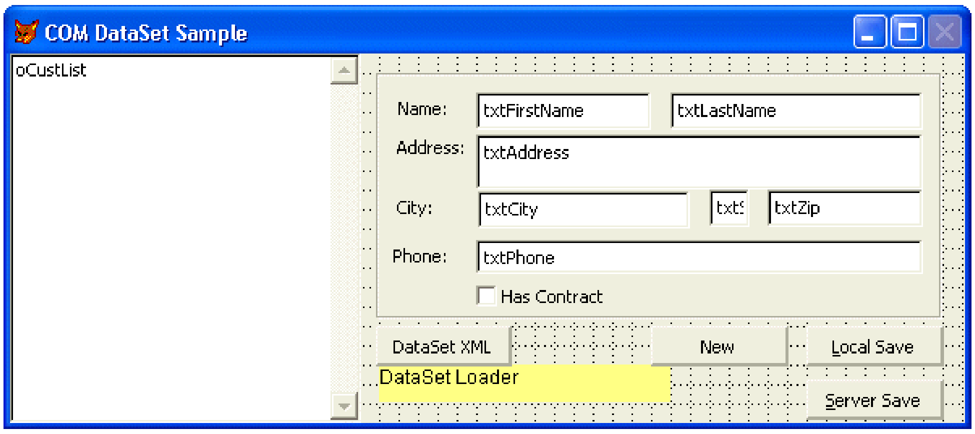


Figure 7 *- The COMAuthors form contains fields bound to a .Net DataSet. All the ControlSource properties are bound to individual fields of a single row in the DataSet.*

The key features of this form are a label control that acts as the DataSet loader (in yellow in Figure 7), and two methods LoadList() and LoadCustomer() that deal with retrieving content from the DataSet.

LoadList() retrieves the DataSet and LoadCustomer then goes out to bind a specific customer to the form. Each of the fields on the form is bound to a field in the current row of the dataset. The Controlsource property points at:

```
THISFORM.oRow.Item(2)
```

Which is the first name of the Authors table in the Pubs database. As you can see you'll have to have specific information about the database schema to know which field you are binding to since we can't access the fields by name in VFP.

To load the data from the DataSet the LoadList() method needs to be called (Listing 24).

Listing 28 (VFP - COMAuthors.scx): Loading the dataset from the .Net COM object

```
*** LoadList
LPARAMETERS llNoServerReload

IF !llNoServerReload
   THIS.oDS = THISFORM.oNet.GetAuthorData("")
   IF THISFORM.oNet.Error
      WAIT WINDOW "Couldn't load data from .NET..." + CHR(13) +;
                  THISFORM.oNet.ErrorMsg
      RETURN
   ENDIF
ENDIF

THIS.oCustList.Clear()

FOR EACH loRow IN THIS.oDs.Tables.item(0).Rows
   THIS.oCustList.AddItem(TRIM(loRow.Item(1)) + ", " + TRIM(loRow.Item(2)))
ENDFOR
```

The .Net method in the COM object that is called was shown in Listing 23 above. The code in the LoadList() method calls the COM method and returns the DataSet, then loops through and accesses each of the First and Last names and sticks them into the list box. When an item in the list box is clicked it calls the LoadCustomer() method to display the individual author (Listing 29).

Lising 29 (VFP – COMAuthors.scx): Loading a single record from the DataSet

```
LPARAMETERS lnIndex

lnIndex = lnIndex - 1

THIS.oRow = THISFORM.oDS.Tables.item(0).Rows.item(lnIndex)
THISFORM.Refresh()
```

This code accesses the DataSet that is already downloaded to the client side so no additional trips to SQL Server are performed at this time. THIS.oRow is updated with the currently selected row of the DataSet and the Form Refresh() simply forces all the fields on the form to refresh their ControlSource values.

This is essentially a disconnected scenario where all manipulation to the DataSet occurs offline. Only the initial pull of the data causes the database to be accessed. All subsequent record reads are from the local copy of the data. The update routines that save and update records of the table also work against the local copy. In fact, changing the content of an existing row in the DataSet is as easy as making a change to the field on the form – since the fields are bound to the recordset's Field objects the changed data is automatically written back into the DataSet everytime a change is made.

Inserting a row into the DataSet requires a little more work. Listing 26 shows the code from the New button event code.

Listing 30 – (VFP – COMAuthors.scx): Inserting a row into the DataSet

```
loRow = THISFORM.oDS.Tables.item(0).NewRow()

*** Fix the empty NULLS
FOR x = 1 TO loRow.Table.Columns.Count
 lcType = LOWER(loRow.Table.Columns.item(x-1).DataType.Name)
 DO CASE
    CASE lcType = "string"
       loRow.Item(x-1) = ""
    CASE lcType = "decimal" OR lctype = "int"
       loRow.item(x-1) = 0
    CASE lcType = "boolean"
       loRow.Item(x-1) = .F.
    CASE lcType = "datetime"
       loRow.Item(x-1) = { : }
 ENDCASE
ENDFOR

lcSoc = INPUTBOX("Please Enter the Social Security number:")

*** Pk Field must be updated!
loRow.Item(0) = lcSoc

THISFORM.oRow = loRow
THISFORM.Refresh()
```

There are a few notes of interest in the code in Listing 25. First off notice that there's a block of code that deals with setting the default values for each field, by walking through the field collection. The reason for this is that the Pubs SQL database is set up with NULLS enabled and thus inserts a new record with all .NULL. values. VFP of course displays these nulls in the fields which is not the effect you desire. Instead you would like to see empty field values. What's interesting in this is that even if you have default values set up for the database the values come down as NULLS, which is a major shortcoming of the DataSet in providing useful data to display.

The code also asks for a Social Security number, that identifies this record in the table and must be unique. It's important to understand that rules are not checked at this point, so if you put in a duplicate value for example, the DataSet will allow this and provide no way to check for this, until the data is saved. The form includes two buttons to save – a local save and a server save.

The local Save operation saves the current record and its changes back into the DataSet and basically deals only with any new rows. Existing rows simply update the fields collection of the selected row and immediately show their data in the DataSet. New rows however create a new DataRow object which is not actually attached to the table until you save. So New does the code in Listing 25 and creates a new oRow object that is freestanding. The Local Save operation then checks for a new record and attaches it to the current DataTable:

```
*** Check for New and Insert Row
IF THISFORM.oRow.RowState = 1
   THISFORM.oRow.Table.Rows.InsertAt(THISFORM.oRow,0)
ENDIF

** Just refresh the list locally
THISFORM.LoadList(.T.)
```

It then calls back into LoadList() to force the list to refresh from the local DataSet. The .T. parameter specifies the list is to be created from the existing DataSet rather than retrieving the data again from SQL Server, which would cause problems since the changes have not been saved to the SQL Server yet.

To save the collectively changed data back into SQL Server the following code is used:

```
IF !ISNULL(THISFORM.oDS)
   llResult = THISFORM.oNET.UpdateAuthorData(THISFORM.oDS)
   IF !llResult OR THISFORM.oNet.Error
      MESSAGEBOX("Error with update" + CHR(13) + THISFORM.oNet.ErrorMsg,;
                 48,"DataSet Update")
   ENDIF
ENDIF
```

This code calls back into the .Net COM object and passes the changed DataSet back to it for updating in .Net. The code for the .Net object's UpdateAuthorData is shown in Listing 31.

Listing 31 (C#): The .Net code to update the DataSet

```
public bool UpdateAuthorData(DataSet loDs)
{
   this.SetError("");

   string cConnection = "server=(local);database=pubs;uid=sa;pwd=";
   SqlConnection oConn = new SqlConnection(cConnection);

   SqlDataAdapter oAdapter = new SqlDataAdapter();
   oAdapter.SelectCommand =
      new SqlCommand("select * from Authors",oConn);

   try
   {
      // *** Create the Update/Insert Commands
      SqlCommandBuilder oCmdBuilder = new SqlCommandBuilder(oAdapter);
      oAdapter.Update(loDs,"Authors");
   }
   catch (Exception ex)
   {
      this.SetError(ex.Message);
      return false;
   }

   return true;
}
```

This code receives the DataSet from VFP via the loDs parameter and then uses standard .Net data access code to call the Update method of the DataAdapter to write the updated data back into the SQL database. It writes the data and returns either true or false depending on success of the Update operation.

Note that you much have to use this indirect approach to updating the database, rather than updating the data directly from within VFP, because VFP does not have direct access to the DataAdapter or CommandBuilder objects. Although you can expose those objects through COM by subclassing them and adding the appropriate export attributes, there are problems with the way some of these objects are returned over COM. It can be made to work, but it's actually much easier and more efficient to use this sort of go between routine to perform the tasks for data retrieval and data storage for you.

Using .Net classes that don't work with COM

Datasets expose some of their functionality but not all of them. Same is true for most other .Net components. Specifically it's very difficult or in many cases simply not supported to access components that expose collection objects.

There are two problems of why this is occurring: First most .Net built-in objects make extensive use of overloaded methods (same methodname – different parameters/return values). Since COM doesn't support overloaded methods only one of the methods in question can be exposed. Usually it'll be the one that has the most parameters which means that calling many of the methods is a lot more difficult in VFP than it is in .Net as you have to specify parameters that are typically left off as 'defaults' in .Net. For example, the DataSet has the ability to access collection items both by name and by index number which is basically an operator overload of the [] expression. But VFP can only use one of them in this case using indexed keys. With other classes – like the Reflection classes for example you simply cannot access the collections at all. In

yet other cases properties and methods simply are not exposed to COM at all.
I recently needed some functionality to read .Net type information in order to import it into HTML Help Builder. Help Builder already supports importing data from COM objects and VFP classes, so adding the ability to also import from .Net components was a useful addition. This is fairly easy with Reflection where in C# you can do something like this:

```
Assembly loAssembly;
loAssembly = Assembly.LoadFrom("d:\temp\someassembly.dll");

foreach (TypeInfo loType in loAssmbly.GetTypes()) {
    MessageBox.Show(loType.Name);
    Foreach (MethodInfo loMethod in loType.GetMethods() {
          MessageBox(loMethod.Name);
    }
}
```

You can basically loop through all the type info including methods, properties, parameters, return values etc. It's very powerful and is a signature of the type system in .Net.

It would have been nice if I could simply return this info to VFP directly by simply creating a COM exposed method that does:

```
public Assembly GetAssembly(string lcFilename) {
  Assembly loAssembly;
  loAssembly = Assembly.LoadFrom(lcFilename);
  return loAssembly;
}
```

You can indeed receive this assembly inside of VFP.

```
loParse = CREATEOBJECT("wwReflection.TypeParser")
loAssembly = oParse.GetAssembly("D:\temp\wwscripting.dll")
loTypes = loAssembly.GetTypes()
? loTypes.Length
? loTypes.item[0].Name
```

You even get Intellisense on the Assembly object returned. But unfortunately there's where the features end. The Types object is not accessible (at least using standard methods for this object) and you can't loop through this data the way you can in .Net. Bummer.

Since I really needed this functionality in my application I decided that I needed to create a wrapper object that's properly exposed via COM. Basically this object creates subobjects that somewhat mimic the format of the reflection classes. In addition, I added functionality to also retrieve documentation information from a C# code documentation file if that is available and parsed that into the same object. I won't divulge this lengthy code here, but I'll show the structure for these objects to give you an idea of how this works:

Listing 31.5 (C#): Wrapper class interface for Reflection classes

```
[ClassInterface(ClassInterfaceType.AutoDual)]
[ProgId("wwReflection.TypeParser")]
public class TypeParser
{
// ** All X properties are native objects
public DotnetObject[] aObjects;
public Type[] aXObjects;
public int nObjectCount = 0;

public string[] aConstants;
public int nConstantCount = 0;

public bool lError = false;
public string cErrorMsg = "";


public string cFilename = "";
public string cXmlFilename = "";

public int GetAllObjects();
public bool ParseXmlDocumentation();
//… implementation not shown
}

[ClassInterface(ClassInterfaceType.AutoDual)]
[ProgId("wwReflection.DotNetObject")]
public class DotnetObject
{
public string cName = "";
public string cScope = "";
public string cHelpText = "";
public string cNamespace = "";

public ObjectMethod[] aMethods;
public int nMethodCount = 0;

public ObjectProperty[] aProperties;
public int nPropertyCount = 0;

public int LoadMethods(TypeInfo loType);
public int LoadProperties(TypeInfo loType);
// … implementation not shown
}

[ClassInterface(ClassInterfaceType.AutoDual)]
[ProgId("wwReflection.ObjectMethod")]
public class ObjectMethod
{
public string cName = "";
public string cParameters = "";
public string cRawParameters = "";
public string cDescriptiveParameters = "";
public string cReturnType = "";
public string cHelpText = "";
public string cScope = "";
public string cOther = "";
public string[] aParameters = null;
public int nParameterCount = 0;

// no methods
}

[ClassInterface(ClassInterfaceType.AutoDual)]
[ProgId("wwReflection.ObjectProperty")]
public class ObjectProperty
{
public string cName = "";
public string cType = "";

public string cScope = "";
public string cDefaultValue = "";

public string cFieldOrProperty = "";

public string cHelpText = "";

// no methods
}
```

You can see that this clipped interface that it uses nested objects and arrays of objects in multiple classes to present a hierarchy similar to what Reflection provides. But these are wrapped in a safe manner for COM – these classes are easily accessible from VFP and are now fully exposed via Intellisense because all objects are exported as COM accessible objects. I can now do:

Listing 31.6 (VFP): Using the wrapper COM class interface

```
loParse = CREATEOBJECT("wwReflection.TypeParser")
COMARRAY(loParse,10)

loParse.cFilename = "D:\temp\wwscripting.dll"
loParse.cXmlFilename = "D:\tempwwscripting.xml"

lnCount =  loParse.GetAllObjects()

loObjects = loParse.aObjects
FOR x = 1 TO lnCount
    LOCAL loObject as wwReflection.DotnetObject
    loObject = loObjects[x]
    ? loObject.cName + loObject.cHelpText

   loMethods = loObject.aMethods
   FOR y = 1 TO  loObject.nMethodCount
       LOCAL loMethod as wwReflection.ObjectMethod
      loMethod = loMethods[y]
      ? " " + loMethod.cScope + " " + ;
        loMethod.cName + "(" + loMethod.cParameters + ") as " +;
        loMethod.cReturnType
      ? "      "  + loMethod.cOther
      ? "      "  + loMethod.cHelpText
   ENDFOR

   loProperties = loObject.aProperties
   FOR y = 1 TO  loObject.nPropertyCount
       LOCAL loProperty as wwReflection.ObjectProperty
      loProperty = loProperties[y]
      ? " " + loProperty.cScope + " " + loProperty.cName + ;
        " as "  + loProperty.cType
      ? " " + loProperty.cHelpText
    ENDFOR
  ENDFOR
```

In this scenario I made the decision that I will put all the required parsing logic into the .Net component rather than letting the Fox component deal with anything. So I did the fixup of the cHelpText property from the XML documentation if available for example as well as parsing parameters and return values into properly formatted strings for display rather than returning arrays of parameters. In short I used the .Net component to perform some of my 'business' logic for my application rather than serving a generic purpose (it's still very usable generically for other applications though).

Obviously there are lots of choices if you create wrapper objects from passing back internal .Net objects that VFP *can* access to creating your own wrapper classes, to creating proxy methods that invoke the appropriate method or property and simply return a value.

It's very possible, but it does require some extra work.

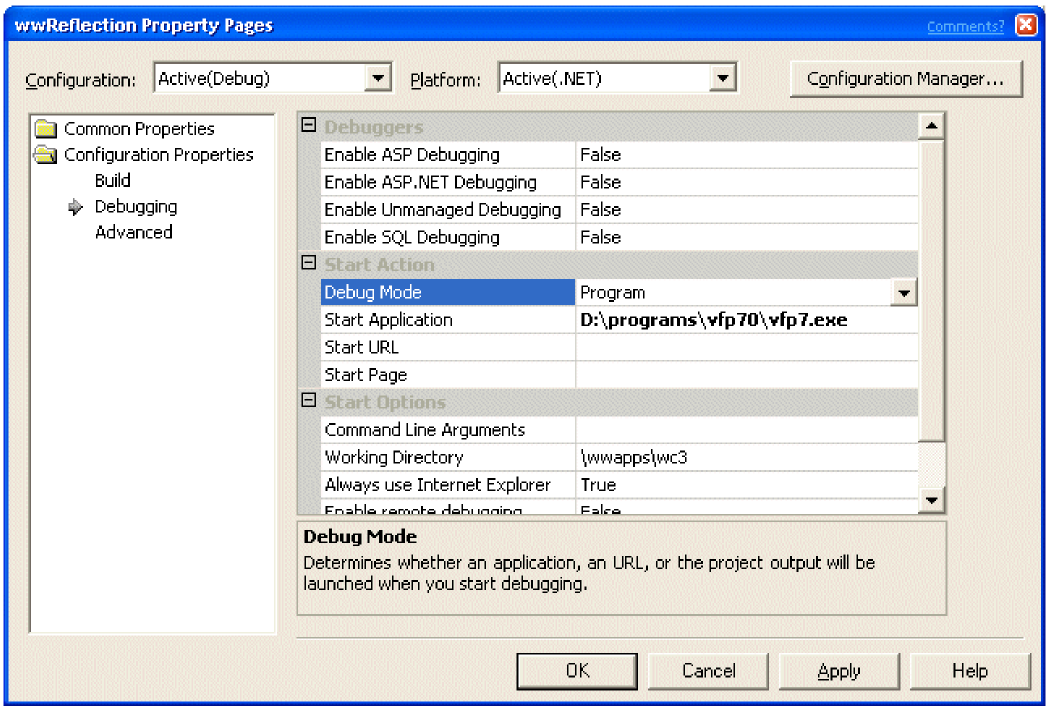
Debugging .Net classes from VFP

You can also debug .Net classes from VFP code by configuring the .Net project. Basically you set up the .Net project that hosts your COM exposes component by having it run VFP as the startup application. Any calls into the COM DLL or Exe can then trigger breakpoints in the .Net code.

To set this up in Visual Studio .Net you can do the following:

1. Select the project that hosts your .Net class exposed via COM in the Project Explorer
2. Right click and select properties
3. Got Configuration Properties | Debugging
4. Set the Debug Mode to Program
5. Set the Start Application to your VFP executable (IDE or your own EXE)
6. Set the Working Directory to where you want VPF to start
7. Set any command line arguments you may need.

Figure 6.1 shows what the dialog looks like in VS.Net.


wwReflection Property Pages

Configuration: Active(Debug)   Platform: Active(.NET)   Configuration Manager...

Common Properties
Configuration Properties
Build
Debugging
Advanced

Debuggers	
Enable ASP Debugging	False
Enable ASP.NET Debugging	False
Enable Unmanaged Debugging	False
Enable SQL Debugging	False
Start Action	
Debug Mode	Program
Start Application	D:\programs\vfp70\vfp7.exe
Start URL	
Start Page	
Start Options	
Command Line Arguments	
Working Directory	\wwapps\wc3
Always use Internet Explorer	True
Enable remote debugging	False

Debug Mode
Determines whether an application, an URL, or the project output will be launched when you start debugging.

***Figure 6.1** – You can debug .Net components with calls made from Visual FoxPro by configuring VFP as the project's startup application. When your VFP code calls the .Net code VS.Net will stop on any breakpoints.*

Once this is configured properly you can now start your project by pressing the Run button (f5) in VS.Net, which will fire up Visual FoxPro or your application. You can create an instance of the .Net COM object make a call on it and the VS.Net debugger will stop on any breakpoints set in the .Net code.

If you've ever tried to debug COM objects called from VFP before, I think you will appreciate how easy this process is compared to not at all being able to debug a COM object by any means. Debugging in .Net in general is a pure joy – simply because any kind of code can be debugged even if it spans multiple projects or as in this case multiple application environments.

Interop with Web Services

The final common mechanism to exchange data between VFP and .NET data involves Web Services. I've left this one for last because I believe that Web Services as a pure Interop technology are not the best mechanism to pass data or logic around. It's slower than the other mechanisms and involves additional configuration and setup that may be overkill for common applications.

However, if the distributed Web Service metaphor makes sense for your application design and you need to pass data over the Web anyway, then Web Services can offer this functionality in a relatively painless matter with easy tools available to publish Web Services from either the unmanaged COM side (VFP + SOAP Toolkit) or from .NET with .NET Web Services.

Using .NET Web Services in Visual FoxPro

I don't want to go into great detail here on how to create a Web Service with the SOAP Toolkit or .NET since these topics have been covered ad nauseum in other articles and publications. For .NET Web Services I'm going to use a Web Service from some previous articles. You can view the article and download the code from: http://www.west-wind.com/presentations/dotnetwebservices/DotNetWebServices.asp.

To call a Web Service with Visual FoxPro you can use the SOAP Toolkit, which is a COM client that can access Web Services and parse the SOAP packets locally. In addition, the Toolkit provides an easy to use proxy that pulls data down easily.

If you're returning simple types (strings, integers, Boolean, DateTime values etc.) using the SOAP Toolkit is as easy as the following code demonstrates (Listing 23).

Listing 26 (VFP): SOAP Client code to retrieve a .NET Web Service method result
```
LOCAL loProxy as MSSOAP.SoapClient

loProxy = CREATEOBJECT("MSSOAP.SoapClient")
loProxy.MsSoapInit("http://localhost/codeservice/codewebservice.asmx?WSDL")

? loProxy.HelloWorld("Rick")
? loProxy.GetServerTime()
? loProxy.ReturnXml("Riæck")
```

This requires that the SOAP Toolkit Version 2.0 is installed on the client. You simply instantiate the SOAP client and point it at the WSDL file that contains the service description. To get a .NET Web Services' WSDL file just access the Web Service's URL with querysting of WSDL (?WSDL), which is dynamically generated from the Web Service's class interface.

The SOAP Toolkit then takes the WSDL content and adds the ability to dynamically call the Web Service's methods through 'simulated' methods of the proxy object. I say simulated because these methods are actually being called internally by the SoapClient class whenever a method that doesn't exist is accessed. The client parses the method name, maps it to the WSDL file schema that describes the method signature and then calls the Web Service method over the wire using the SOAP XML protocol.

The good news is that this is very easy to accomplish. The bad news is that if something goes wrong things can be tricky to troubleshoot as the simple Soap Proxy client is fairly limited. If there's a connection problem or an error that's not directly returned from the SOAP Server the client application will blow up with an unspecified error. For this reason the Soap classes also include a lower level interface which requires a bit more code to run the equivalent code basically allowing you to access the SOAP XML content directly and parsing it.

Errors thrown by the server are forwarded to the simple SoapClient as COM exceptions. The error message of the original COM error is thrown forward and the client can capture this error with the VFP error handlers. A good way to write safe Web Service client code is to use wrapper classes that wrap the proxy method calls to avoid blowing up in code and retrieving the error. I use an Eval class for this:

```
loEval = CREATEOBJECT("wwEval")
lvResult =  loEval.Evaluate([loProxy.CrashMe()])
IF loEval.lError
   ? loEval.cErrorMsg
ELSE
   ? lvResult
ENDIF
```

Unfortunately .NET throws some rather verbose error messages. For example the following error message results from an invalid connection used on the server (ie. a 'hard' exception):

Listing 27 (C#): A Web Service method that forces an Exception
```
[WebMethod]
public string CrashMe()
{
   // *** Simulate an error
   Exception ex = new Exception("Invalid Connection String");
   throw(ex);

   return "";
}
```

This results in the following error message:

```
System.Web.Services.Protocols.SoapException: Server was unable to process request. --->
System.Exception: Invalid Connection String
   at CodeService.CodeWebService.CrashMe() in d:\westwind\codeservice
\codewebservice.asmx.cs:line 164
   --- End of inner exception stack trace ---
```

Not much you can do with this error message other than log it and let the user know the call failed.

Complex Types: Retrieving Objects from .NET

So far I've shown you how to retrieve simple types which is pretty straight forward as the SOAP Toolkit can automatically convert these types and pass them back as you would expect. If you want to return complex parameters such as objects or DataSets from .NET you will have to do a little more work since the SOAP toolkit cannot directly deal with them. It actually offers two methods that you can use – accessing the XML directly or using the low level objects to construct objects to be loaded from the XML. Frankly the latter approach takes a fair amount of code and requires updating whenever the object structure changes that it isn't really worth working with. In this situation it's actually easier to use the XML directly instead.

The SOAP toolkit returns complex type results as XMLDOM Nodelists. A NodeList is a collection of DOM nodes. It seems to me like a really odd choice to return the NodeList as opposed to the parent node of the list, but that's the way the toolkit works. Consider the following Web Service method:

Listing 28 (C#): A Web Service method that returns a nested object

```
[WebMethod]
public Customer GetCustomerByName(string lcName)
{
   Customer oCustomer = new Customer();

   if (!oCustomer.Load(lcName))
      return null;

   return oCustomer;
}
```

This method returns a Customer object that contains customer data as well as an address sub object. This returns an object embedded into the SOAP response like this:

```
<GetCustomerResult xmlns="http://tempuri.org/">
  <cLastName>Strahl</cLastName>
  <cFirstName>Rick</cFirstName>
  <cCompany>West Wind Technologies</cCompany>
  <tEntered>2000-10-01T00:00:00.0000000-10:00</tEntered>
  <lError>false</lError>
  <cErrorMsg />
  <oAddress>
     <cAddress>32 Kaiea Place</cAddress>
     <cPhone>(808) 579-8342</cPhone>
     <cEmail>rstrahl@west-wind.com</cEmail>
  </oAddress>
</GetCustomerResult>
```

Note that object results are returned as embedded XML streams rather than XML strings which would be encoded (ie. the < > tags would be marked with &lt; and &gt;).

To retrieve this XML from via SOAP you can use the following VFP code (Listing 29).

Listing 29 (VFP): Retrieving an XML string from a Complex .NET return type

```
loProxy = CREATEOBJECT("MSSOAP.SoapClient")
loProxy.MsSoapInit("http://localhost/codeservice/codewebservice.asmx?WSDL")

loCustomerXMLNodeList = loProxy.GetCustomer(1)

IF loCustomerXMLNodeList.Length > 0
lcXML = loCustomerXMLNodeList.Item(0).parentnode.xml
    ShowXML(lcXML)   && Display the XML
ELSE
    ? "No data returned"
ENDIF
```

Ok, this gives us an XML string – what to do with that? Well, VFP provides no native mechanism to parse this data back into an object. The SOAP Toolkit provides some helper methods that can help you map this XML back to an object using the ISoapTypeMapper interface, but frankly it requires even more code than using the DOM directly to parse this data and also requires that the object being mapped exists.

The biggest issue in retrieving the data is to format the contained types properly. wwXML can help with this and the simple code to do this looks something like this. First a class must be created on the client to hold the actual data (Listing 30).

Listing 30 (VFP): Class definition to hold the complex type

```
DEFINE CLASS Customer as Session

cLastName = ""
```

```
cFirstName = ""
cCompany = ""
tEntered = { : }
lError = .F.
cErrorMsg = ""
oAddress = null

FUNCTION Init
THIS.oAddress = CREATEOBJECT("Address")
RETURN

ENDDEFINE

DEFINE CLASS ADDRESS as Session

cAddress = ""
cPhone = ""
cEmail = ""

ENDDEFINE
```

The class can then be created and the values retrieve from the DOM like this (Listing 31).

Listing 31 (VFP): Reading an object in VFP code using wwXML

```
loProxy = CREATEOBJECT("MSSOAP.SoapClient")
loProxy.MsSoapInit("http://localhost/codeservice/codewebservice.asmx?WSDL")

loCustomerXMLNodeList = loProxy.GetCustomer(1)
lcXML = loCustomerXMLNodeList.Item(0).parentnode.xml

*** Now manually parse the object
loXML = CREATEOBJECT("wwXML")
loDOM = loXML.LoadXML(lcXML)    && Load MSXML DOM document
loRoot = loDOM.DocumentElement

loCustomer = CREATEOBJECT("Customer")
loCustomer.cCompany = loXML.XPathValueToFoxValue(loRoot,"cCompany")
loCustomer.cFirstName = loXML.XPathValueToFoxValue(loRoot,"cFirstName")

loCustomer.tEntered = loXML.XPathValueToFoxValue(loRoot,"tEntered","datetime")
loCustomer.lError = loXML.XPathValueToFoxValue(loRoot,"lError","boolean")

? loCustomer.cCompany
? loCustomer.tEntered
? loCustomer.lError

loCustomer.oAddress.cAddress =
                 loXML.XPathValueToFoxValue(loRoot,"oAddress/cAddress")
loCustomer.oAddress.cPhone =
                 loXML.XPathValueToFoxValue(loRoot,"oAddress/cPhone")

? loCustomer.oAddress.cAddress
? loCustomer.oAddress.cPhone
```

As you can see this process is not exactly automatic. The key thing is to retrieve the XML which is accomplished with:

```
lcXML = loCustomerXMLNodeList.Item(0).parentnode.xml
```

The SOAP Toolkit returns an XMLNodeList which ideally you should check for size first (.Length > 0). You can then walk up to the parent node to retrieve the entire XML or other wise parse the XML fragment using the DOM.

I'm using wwXML and some of its helper methods to read the values from the DOM and properly type and assign them to the VFP object. The helper method XPathValueToFoxValue() uses an XPath query to retrieve the text and then convert it into the proper type if specified. If not specified the string value of the element is returned. The first parameter is the root node from which the XPath query is executed which is

```
<GetCustomerResult xmlns="http://tempuri.org/">
```

in the object XML or loDOM.DocumentElement. The method then basically does a SelectSingleNode() behind the scenes to retrieve the values and calls the internal XmlValueToFoxValue() method to convert the type if specified. As you can see the Boolean and DateTime values are properly typed after this code runs.

You might think that it looks like you could automate this process, right? After all you know what the type information of the class is. In fact, wwXML includes a method called XMLToObject which could possibly import this object directly. Unfortunately, there's a problem with type character casing. .NET returns types in type sensitive format (note the property names are tEntered, cCompany etc.) which is a problem. There's no way for wwXML to figure out what the proper casing for the properties is and since XML is case sensitive when using XPATH queries and AMEMBERS always returns UPPER case property names, there's no way to match property casing to the XML casing.

If you chose to use only lower case property names in the .NET class published via the Web Service, wwXML::XMLToObject would work, but as it stands the longer hand syntax used here is required. The code above should be stored into a method of the business object most likely as a type mapper and updated as the classes change.

More Complex Types: DataSets

Another common type that might be returned from a .NET Web Service is a DataSet or an updategram (which is really just a specialized DataSet with just the changes). I previously showed how you can utilize wwXML to convert XML from a DataSet into a VFP cursor. Listing 32 shows how to do it with the results from a Web Service.

Listing 32 (VFP): Retrieving a DataSet into a cursor

```
loProxy = CREATEOBJECT("MSSOAP.SoapClient")
loProxy.MsSoapInit("http://localhost/codeservice/codewebservice.asmx?WSDL")

loAuthorDsXML = loProxy.GetAuthorData("")
IF loAuthorDsXml.Length > 0
  lcXML = loAuthorDsXML.item(0).parentNode.xml
  ShowXML(lcXML)
ELSE
  ? "No data returned in dataset"
  return
ENDIF

*** Must precreate the cursor
CREATE CURSOR TAuthors (au_id c(12), au_lname c(20), au_fname c(20),;
                        phone c(20), ADdress M,State c(2),Zip c(8),;
                        contract L, pk I)

loXMl = CREATEOBJECT("wwXML")
loXML.DataSetXMLToCursor(lcXML,"TAuthors","Authors")

BROWSE
```

The pre-requisite here is that you are required to precreate the cursor so that wwXML can figure out which fields to retrieve from the XML stream and what type to convert them to. It's also required in order to figure out the field lengths which are not provided as part of the dataset.

What about going the other way? Sending data to XML from the client to the server? The easiest way to do this is to use VFP's native CURSORTOXML function and send the result back to the Web Service as a string. Then use import the cursor using the DataSet's ReadXml() method as described earlier in the ASP section. As of right now wwXML doesn't include a mechanism to create a DataSet directly from data – this is mainly because ReadXml() serves the import functionality just fine. A future version will provide this functionality. A future version of VFP hopefully too will provide more direct support for DataSets to facilitate the process of passing data back and forth more easily via XML.

Keep in mind that the facilities I've described here for DataSet conversion into cursors is very minimal and doesn't take advantage of some of the cool features that DataSet provide including auto-updates and persistence. It is possible to build this type of functionality by building a translation layer for the Fox data semantics of data updates (TableUpdate() and XML Diff Grams for example) and map them to what .NET expects to provide some semblance of full DataSet compatibility. But this process would be a lot of work and given the data model differences probably not worth the effort. If you really need all the functionality DataSets provide, it's probably a good idea to start thinking about using .NET native code to build that functionality, or at the very least built wrappers that are more easily callable from the VFP client utilizing the functionality of .NET rather than re-implementing the functionality in VFP. I think this is a common theme throughout these interop technologies I've described in this article.

Interop Summary

Phew – is your head spinning with all this information? There's a lot of info crammed into this one article, and I have a feeling it will only get bigger over time. In fact some of the topics will probably break out into separate articles with even more detail for each technology.

I hope this document gives you a pretty good idea on where to start with your .NET VFP integration tasks and some tools and ideas that give you a head start on providing useful functionality for applications that need to use both .NET and VFP technologies in tandem.

.NET provides the tools to make this integration relative painless, but realize that the mechanisms described here are meant mainly for backwards compatibility with existing technology rather than technology that's moving forward. Microsoft is definitely betting on .NET for the future, and the Interop mechanisms I described here are one way to provide a link with the existing code base that exists today. Interop gives you the opportunity to ease into .NET without having to start totally from scratch and being able to support both the old and the new simultaneously.

Just keep in mind that there's a penalty involved for this – the overhead of .NET, the installation issues, versioning and security. So, just because the technology is there it doesn't mean that you should immediately run out and integrate it. Make sure it fits your needs and especially fits your business, development and installation requirements before jumping in. But you can be assured that when the decision is made the tools are there for you to integrate VFP and .NET.

Resources

Code for this article:
http://www.west-wind.com/presentations/VfpDotNetInterop/VfpDotNetInterop.zip

Rick Strahl is president of West Wind Technologies on Maui, Hawaii. The company specializes in Web and distributed application development and tools with focus on Windows 2000, ISAPI, Visual FoxPro, .NET and Visual Studio. Rick is author of West Wind Web Connection, a powerful and widely used Web application framework for Visual FoxPro and West Wind HTML Help Builder. He's also a Microsoft Most Valuable Professional, and a frequent contributor to magazines and books. He is co-publisher and co-editor of CoDe magazine, and his book, "Internet Applications with Visual FoxPro 6.0", is published by Hentzenwerke Publishing. For more information please visit: http://www.west-wind.com/.