Session E-CALL

Calling .NET Code from VFP the Easy Way

Doug Hennig
Stonefield Software Inc.
Email: dhennig@stonefield.com
Corporate Web site: www.stonefieldquery.com
Personal Web site : www.DougHennig.com
Blog: DougHennig.BlogSpot.com
Twitter: DougHennig

Overview

At the German DevCon 2011, Doug's "Creating ActiveX Controls for VFP Using .NET" session showed how to create .NET components that can be used in Visual FoxPro applications. However, these types of controls suffer from a couple of issues: they have to be registered for COM on the customer's system and there are limitations in working with .NET Interop in VFP that prevent many things from working correctly. This session shows how Rick Strahl's wwDotNetBridge eliminates these issues and provides some practical examples of how this tool can be used in your applications.

Introduction

The Microsoft .NET framework has a lot of powerful features that aren't available in VFP. For example, dealing with Web Services is really ugly from VFP but is simple in .NET. .NET also provides access to most operating system functions, including functions added in newer version of the OS. While these functions are also available using the Win32 API, many of them can't be called from VFP because they require callbacks and other features VFP doesn't support, and accessing this functions via .NET is easier anyway.

Fortunately, there are various mechanisms that allow you to access .NET code from VFP applications. For example, at the German DevCon 2011, my "Creating ActiveX Controls for VFP Using .NET" session showed how to create .NET components that can be used in VFP applications. However, these types of controls suffer from a couple of issues: they have to be registered for COM on the customer's system and there are limitations in working with .NET Interop in VFP that prevent many things from working correctly.

Recently, Rick Strahl released an open source project called wwDotNetBridge. You can read about this project on his blog (http://tinyurl.com/cgj63yk). wwDotNetBridge provides an easy way to call .NET code from VFP. It eliminates all of the COM issues because it loads the .NET runtime host into VFP and runs the .NET code from there. I strongly recommend reading Rick's blog post and white paper to learn more about wwDotNetBridge and how it works.

.NET COM interop

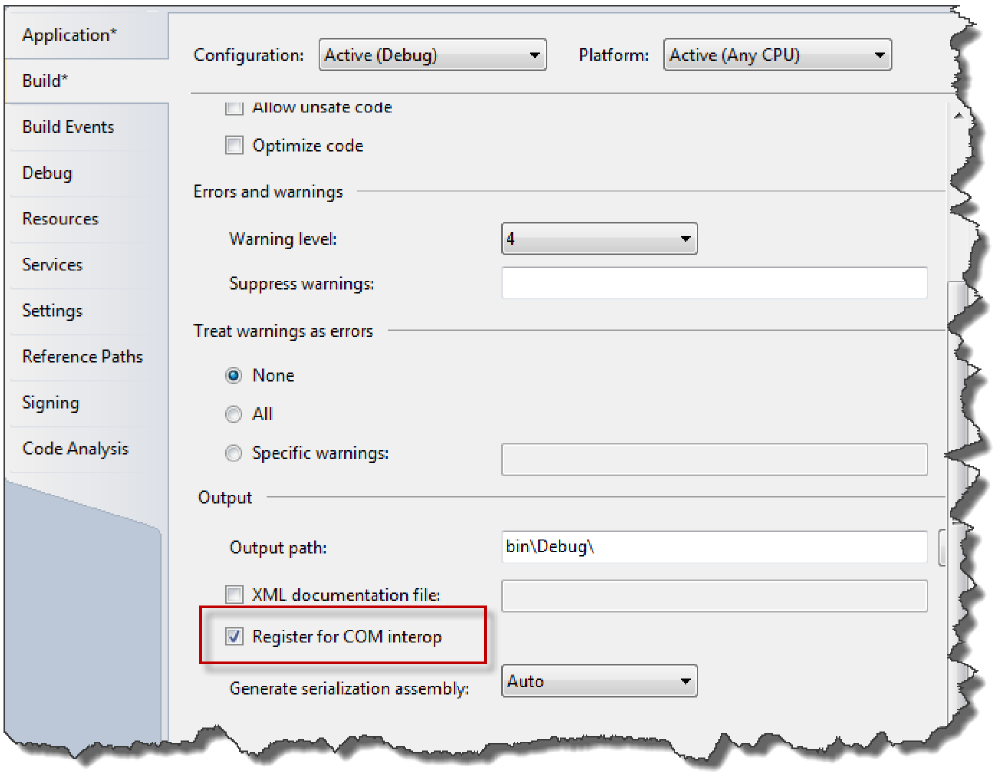
Let's start with an overview of how .NET supports COM interoperability. .NET components by default can't be accessed from a COM client, but you can turn this on by adding some attributes to the .NET class and registering the resulting DLL using the RegAsm.exe utility that comes with the .NET framework. RegAsm is like RegSvr32, which you may know is used to register COM objects, but is used to register .NET assemblies for COM.) Of course, this means you need access to the source code for the .NET component, which is fine if it was written in-house but won't be the case for a native .NET class or a third-party component.

Let's create a simple class to see how .NET COM interop works. Start Microsoft Visual Studio (VS) as an administrator; if you don't, you'll get an error later when VS tries to register the component we'll build as a COM object. Create a new project and choose the "Class Library" template from the C# templates. Let's call the project "InteropSample." Put the code shown in **Listing 1** into the default Class1.cs.

Listing 1. The code for Class1.cs.

```
using System.Collections.Generic;
using System.Runtime.InteropServices;
using System.Linq;

namespace InteropSample
{
    [ComVisible(true)]
    [ClassInterface(ClassInterfaceType.AutoDual)]
    public class Class1
    {
        public string HelloWorld(string name)
        {
            return "Hello, " + name;
        }
    }
}
```

Although we could manually use RegAsm to register this class as a COM component, there's an easier way. Select the project in the Solution Explorer, right-click, and choose Properties. In the Build page, turn on "Register for COM interop" (**Figure 1**). Of course, this only works on your system; you have to use RegAsm to register it on another machine.



Configuration: Active (Debug)
Platform: Active (Any CPU)

- [ ] Allow unsafe code
- [ ] Optimize code

Errors and warnings

Warning level: 4

Suppress warnings:

Treat warnings as errors

- (●) None
- ( ) All
- ( ) Specific warnings:

Output

Output path: bin\Debug\

- [ ] XML documentation file:
- [✓] Register for COM interop

Generate serialization assembly: Auto

Figure 1. Turn on "Register for COM interop" to automatically register the control on your system.

Build a DLL by choosing Build Solution from the Build menu.

Let's try it out. Start VFP and type the following in the Command window:

```
loClass = createobject('InteropSample.Class1')
messagebox(loClass.HelloWorld('Doug'))
```

You should see "Hello, Doug" in a window. Pretty easy, right? Let's take a look at a more complicated example. Add the code in **Listing 2** to the Class1 class and the code in **Listing 3** to the end of Class1.cs (before the final closing curly brace).

Listing 2. Add this code to the existing code in the Class1 class.

```
public List<Person> People = new List<Person>();

public Person AddPerson(string firstName, string lastName)
{
   Person person = new Person();
   person.FirstName = firstName;
   person.LastName = lastName;
   People.Add(person);
   return person;
}

public Person GetPerson(string lastName)
{
   Person person = People.Where(p => p.LastName == lastName)
      .FirstOrDefault();
   return person;
}
```

Listing 3. Add this code to the end of Class1.cs.

```
[ComVisible(true)]
[ClassInterface(ClassInterfaceType.AutoDual)]
public class Person
{
   public string FirstName { get; set; }
```

```
        public string LastName { get; set; }
    }
```

Close VFP; we need to do that when we make changes and rebuild our .NET project because VFP holds a reference to the COM object which prevents the build from succeeding. Build the solution, then start VFP and type the following in the Command window:

```
loClass = createobject('InteropSample.Class1')
loClass.AddPerson('Doug', 'Hennig')
loClass.AddPerson('Rick', 'Schummer')
loClass.AddPerson('Tamar', 'Granor')
loPerson = loClass.GetPerson('Granor')
messagebox(loPerson.FirstName + ' ' + loPerson.LastName)
```

Everything is still working as expected. Now try this:

```
messagebox(loClass.People.Count)
```

First, notice that although there's a People member of Class1, it doesn't show up in VFP IntelliSense. Second, you'll get an OLE "Not enough storage is available to complete this operation" error when you execute this command. The reason for both of those is that People is of type List<Person>, which is a .NET generic type. Generics aren't available to COM clients. That's a huge limitation because generics are used a lot in .NET classes.

Here are some other commonly-used .NET things that aren't available through COM interop:

- Value types
- Structures
- Enumerations (also known as "enums")
- Static methods and properties
- Guids

There are other problems as well.

- Arrays are marshaled by value into VFP as VFP arrays rather than .NET arrays, so they lose some functionality and changes to the array aren't reflected back in the .NET copy.
- COM doesn't support constructors (like the Init of a VFP class) that accept parameters.

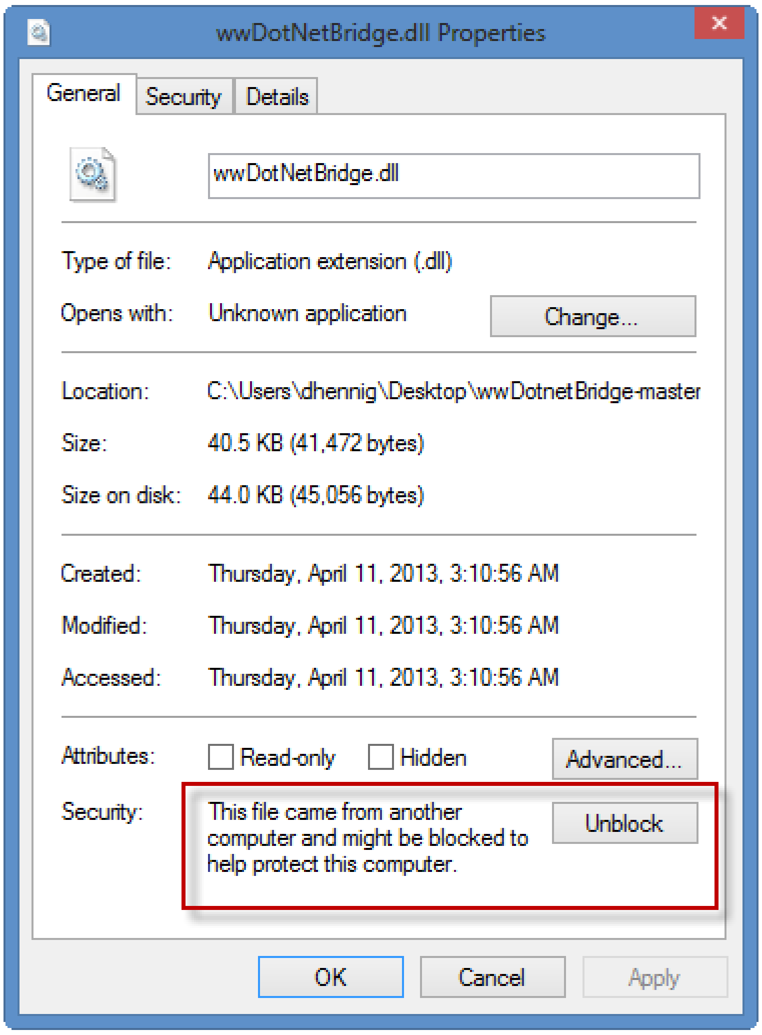
Fortunately, these are all issues wwDotNetBridge can easily handle for us. Let's check it out.

Getting wwDotNetBridge

The first thing to do is download wwDotNetBridge from GitHub: http://tinyurl.com/ce9trsm. If you're using Git (open source version control software), you can clone the repository. Otherwise, just click the "Download ZIP" button on that page to download wwDotnetBridge-master.ZIP. Unzip this file to access all of the source code or just pull out the following files from the Distribution folder:

- ClrHost.DLL
- wwDotNetBridge.DLL
- wwDotNetBridge.PRG

Note that since wwDotNetBridge.DLL is downloaded, you'll likely have to unblock it to prevent an "unable to load Clr instance" error when using wwDotNetBridge. Right-click the DLL, choose Properties, and click the Unblock button shown in **Figure 2**.


wwDotNetBridge.dll Properties

General | Security | Details

wwDotNetBridge.dll

Type of file: Application extension (.dll)

Opens with: Unknown application

Location: C:\Users\dhennig\Desktop\wwDotnetBridge-master

Size: 40.5 KB (41,472 bytes)

Size on disk: 44.0 KB (45,056 bytes)

Created: Thursday, April 11, 2013, 3:10:56 AM

Modified: Thursday, April 11, 2013, 3:10:56 AM

Accessed: Thursday, April 11, 2013, 3:10:56 AM

Attributes: Read-only Hidden

Security: This file came from another computer and might be blocked to help protect this computer.

Figure 2. Unblock wwDotNetBridge.DLL to prevent errors when using it.

Using wwDotNetBridge

Start by instantiating the wwDotNetBridge wrapper class using code like:

```
loBridge = newobject('wwDotNetBridge', 'wwDotNetBridge.prg', '', 'V4')
```

The last parameter tells wwDotNetBridge which version of the .NET runtime to load. By default, it loads version 2.0; this example specifies version 4.0. Note that you can only load one version at a time and it can't be unloaded without exiting VFP. That's why Rick recommends instantiating wwDotNetBridge into a global variable in your applications and using that global variable everywhere you want to use wwDotNetBridge.

The next thing you'll likely do is load a custom .NET assembly (you don't have to do this if you're going to call code in the .NET base library) and instantiate a .NET class. For example, this code loads the InteropSample assembly we were working with earlier and instantiates the Class1 class which lives in the InteropSample namespace:

```
loBridge.LoadAssembly('InteropSample.dll')
loClass = loBridge.CreateInstance('InteropSample.Class1')
```

Note that you need to specify the correct path for the DLL (for example, "InteropSample\InteropSample\bin\Debug\InteropSample.dll" if the current folder is the Samples folder included with the sample files for this document) and you should check the lError and cErrorMsg properties of wwDotNetBridge to ensure everything worked.

Now you can access properties and call methods of the .NET class. The following code is the same as we used earlier:

```
loClass.AddPerson('Doug', 'Hennig')
loClass.AddPerson('Rick', 'Schummer')
loClass.AddPerson('Tamar', 'Granor')
loPerson = loClass.GetPerson('Granor')
messagebox(loPerson.FirstName + ' ' + loPerson.LastName)
```

However, we still can’t access the People member without getting an error for the same reasons we saw earlier. In that case, use the GetPropertyEx method of wwDotNetBridge:

```
loPeople = loBridge.GetPropertyEx(loClass, 'People')
messagebox(loBridge.GetPropertyEx(loPeople, 'Count'))
loPerson = loBridge.GetPropertyEx(loClass, 'People[1]')
messagebox(loPerson.FirstName + ' ' + loPerson.LastName)
```

Here’s another way to access People: convert it to an array by calling CreateArray and then FromEnumerable:

```
loPeople = loBridge.CreateArray()
loPeople.FromEnumerable(loClass.People)
lcPeople = ''
for lnI = 0 to loPeople.Count - 1
   loPerson = loPeople.Item[lnI]
   lcPeople = lcPeople + loPerson.FirstName + ' ' + ;
      loPerson.LastName + chr(13)
next lnI
messagebox(lcPeople)
```

To set the value of a property, call SetPropertyEx. For a static property, use GetStaticProperty and SetStaticProperty:

```
loBridge.GetStaticProperty('MyNameSpace.MyClass', 'SomeProperty')
loBridge.SetStaticProperty('MyNameSpace.MyClass', 'SomeProperty', ;
   'SomeValue')
```

To call a method that can’t be accessed directly, use InvokeMethod or InvokeStaticMethod:

```
loBridge.InvokeMethod(loClass, 'SomeMethod')
loBridge.InvokeStaticMethod('MyNameSpace.MyClass', 'SomeStaticMethod')
```

For example, here’s a call to a static method that returns .T. if you’re connected to a network:

```
llConnected = loBridge.InvokeStaticMethod( ;
   'System.Net.NetworkInformation.NetworkInterface',
   'GetIsNetworkAvailable')
```

Note that no COM registration is required. To demonstrate this, quit VFP and comment out the two sets of COM attributes in Class1.cs:

```
//[ComVisible(true)]
//[ClassInterface(ClassInterfaceType.AutoDual)]
public class Class1
...
//[ComVisible(true)]
//[ClassInterface(ClassInterfaceType.AutoDual)]
public class Person
```

Build the solution again, start VFP, and type:

```
loClass = createobject('InteropSample.Class1')
```

You’ll get a “Class definition INTEROPSAMPLE.CLASS1 is not found” error. However, the wwDotNetBridge commands work just fine. This is actually a very important reason to use wwDotNetBridge; as I mentioned earlier, you can access just about any .NET component, whether in the .NET framework or a third-party assembly, without having to add COM attributes to the source code and rebuilding the component. It also makes deployment easier: no need to use RegAsm to register the assembly on another computer.

How wwDotNetBridge works

wwDotNetBridge consists of the following components:

- wwDotNetBridge.PRG: a program containing the wwDotNetBridge class. This class mostly wraps the methods in the .NET wwDotNetBridge class in wwDotNetBridge.DLL, but also loads the .NET runtime contained in ClrHost.dll.
- wwDotNetBridge.DLL: a .NET DLL that handles all of the interop stuff.
- ClrHost.DLL: a custom version of the .NET runtime host.

The architecture of wwDotNetBridge is shown in **Figure 3**, an updated version of the diagram that appears in Rick's documentation.

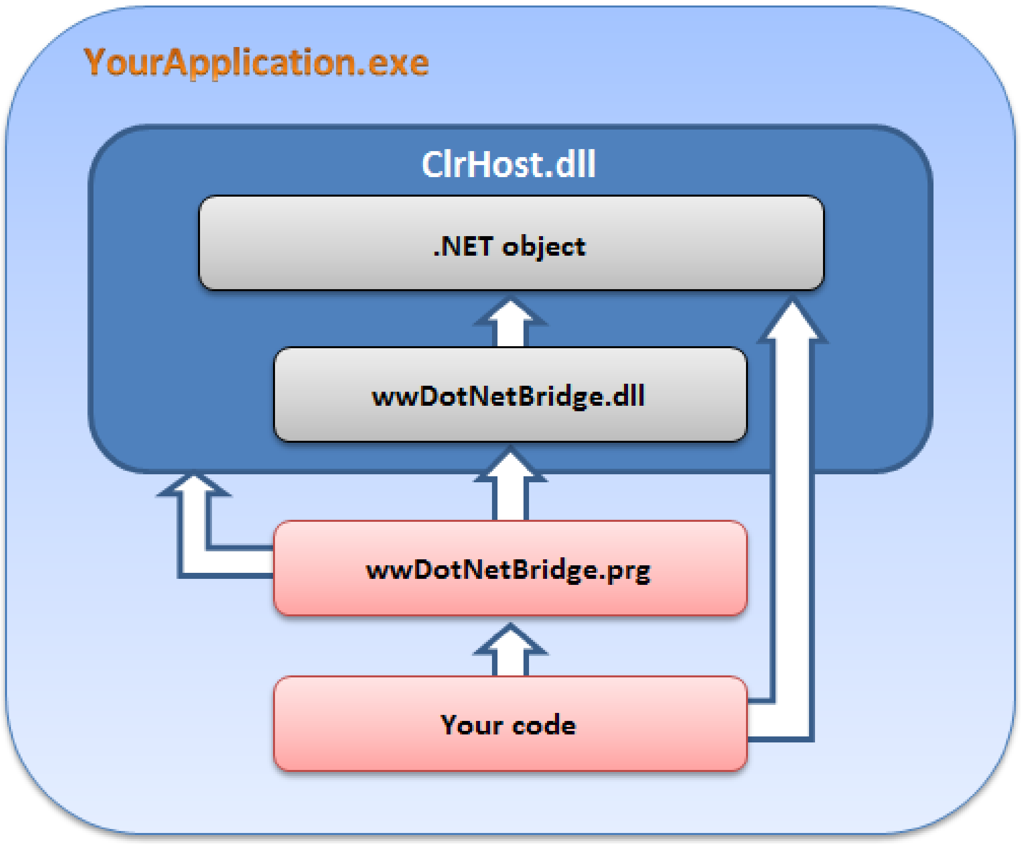


Figure 3. The architecture of wwDotNetBridge.

The first time you instantiate the wwDotNetBridge class in wwDotNetBridge.prg, it loads ClrHost. ClrHost loads wwDotNetBridge.dll, creates an instance of the wwDotNetBridge class in that DLL, and returns that instance to the calling code as a COM object, stored in the oDotNetBridge member of the VFP wwDotNetBridge class. When you call a method of the wwDotNetBridge wrapper class, such as GetPropertyEx, it calls an equivalent method of the .NET wwDotNetBridge to do the actual work. For example, the GetPropertyEx method has this simple code:

```
FUNCTION GetPropertyEx(loInstance,lcProperty)
RETURN this.oDotNetBridge.GetPropertyEx(loInstance, lcProperty)
```

Your code may also call methods or access properties of a .NET object directly once you've created an instance of it using CreateInstance.

Now that we covered the basics, let's look at some practical examples.

Example #1: sending email

There are lots of ways to send email from VFP, all of them using external components since VFP doesn't natively support that. However, not all of them support using Secure Sockets Layer, or SSL, or sending emails formatted as HTML. Since .NET supports both of those, it's an easy matter to use wwDotNetBridge to provide that capability to VFP.

One of the examples in the Examples folder of the wwDotNetBridge downloads shows how to do this. **Listing 4** shows a simplified example.

Listing 4. Sending an email is easy using wwDotNetBridge.

```
loBridge = newobject('wwDotNetBridge', 'wwDotNetBridge.prg', '', 'V4')
loBridge.LoadAssembly('InteropExamples.dll')
loSmtp = loBridge.CreateInstance('Westwind.wwSmtp')

loSmtp.MailServer  = 'myserver.server.net'
loSmtp.UseSsl      = .T.
loSmtp.Username    = 'itsme@whatever.com'
loSmtp.Password    = 'mypassword'
loSmtp.Recipient   = 'whoever@whereever.com'
loSmtp.SenderEmail = 'itsme@whatever.com'
loSmtp.Subject     = 'Email subject'
loSmtp.Message     = 'Body of the message'

loSmtp.SendMail()
```

You can also call SendMailAsync to send the message asynchronously. See Rick's documentation for examples of how to handle events in that case.

To send an HTML-formatted message, set loSmtp.ContentType to "text/html" and Message to the HTML content.

Example #2: XML processing

While VFP is normally very fast at string handling, one thing that's very slow is converting XML into a cursor, using either XMLToCursor() or the XMLAdapter class, when there are a lot of records in the XML. For example, People.XML, included with the samples files for this document, contains 64,000 names and addresses. It takes a whopping 995 seconds, or more than 16 minutes, to convert it to a VFP cursor using XMLToCursor(). (Run TestXMLToCursor.PRG to see for yourself.) Let's see how wwDotNetBridge can help.

One of the things I've learned over the past few years is that .NET's XML parser is very fast. It just takes a few lines of code to read an XML file into a DataSet. If you aren't familiar with a DataSet, it's like an in-memory database consisting of one or more DataTables. Each DataTable has a Columns collection providing information about the columns, such as name, data type, and size, and a Rows collection that contains the actual data. wwDotNetBridge.PRG has a method called DataSetToCursors that takes a DataSet returned from some .NET code and converts it to one or more VFP cursors. However, in testing, I found it to be very slow as well. Looking at the code, it was obvious why: DataSetToCursors uses XMLAdapter, which is what we're trying to get away from.

I decided to try a different approach: have .NET read the XML into a DataSet and have the VFP code create a cursor with the same structure as the first DataTable in the DataSet (which we can get by going through the Columns collection) and fill the cursor with the contents of each object in the Rows collection.

Listing 5 shows the code for the Samples class in Samples.CS. It has a single GetDataSetFromXML method that, when passed the filename for an XML file, reads that file into a DataSet and returns the DataSet. If something goes wrong, such as the XML not being suitable for a DataSet, ErrorMessage contains the text of the error.

Listing 5. The Samples class loads an XML file into a DataSet.

```
public class Samples
{
  public string ErrorMessage { get; private set; }

  public DataSet GetDataSetFromXML(string path)
  {
    DataSet ds = new DataSet();
    FileStream fsReadXml = new FileStream(path, FileMode.Open);
    try
    {
        ds.ReadXml(fsReadXml);
      }
      catch (Exception ex)
      {
        ErrorMessage = ex.Message;
      }
      finally
      {
        fsReadXml.Close();
      }
      return ds;
    }
  }
```

Listing 6 shows the VFP code, taken from TestXML.PRG, that uses the C# class to do the conversion of the XML into a DataSet, then calls the CreateCursorFromDataTable function to use the approach I mentioned earlier to create a VFP cursor from the first DataTable in the DataSet.

Listing 6. TestXML.PRG loads the DataSet returned from the .NET class into a cursor.

```
local lnStart, ;
  loBridge, ;
  loFox, ;
  loDS, ;
  lnEnd1, ;
  loTable, ;
  lnEnd2

* Save the starting time.

lnStart = seconds()

* Create the wwDotNetBridge object.

loBridge = newobject('wwDotNetBridge', 'wwDotNetBridge.prg', '', 'V4')

* Load our assembly and instantiate the Samples class.

loBridge.LoadAssembly('Samples.dll')
loFox = loBridge.CreateInstance('Samples.Samples')

* Get a DataSet from the People.XML file, get the first table, and
* convert it to a cursor.

loDS    = loFox.GetDataSetFromXML('people.xml')
lnEnd1  = seconds()
loTable = loBridge.GetPropertyEx(loDS, 'Tables[0]')
CreateCursorFromDataTable(loBridge, loTable, 'TEMP')

* Display the elapsed time and browse the cursor.

lnEnd2 = seconds()
messagebox(transform(lnEnd1 - lnStart) + ' seconds to create a ' + ;
  'DataSet from the XML and ' + transform(lnEnd2 - lnEnd1) + ;
  ' seconds to create a cursor, for a total of ' + ;
  transform(lnEnd2 - lnStart) + ' seconds.')
browse


function CreateCursorFromDataTable(toBridge, toTable, tcCursor, ;
   tnRecords)
lparameters toBridge, ;
   toTable, ;
   tcCursor, ;
   tnRecords
```

```
local lnColumns, ;
   lcCursor, ;
   laColumns[1], ;
   laMaxLength[1], ;
   lnI, ;
   loColumn, ;
   lcColumnName, ;
   lcDataType, ;
   lcType, ;
   lcAlias, ;
   lnRecords, ;
   lnRows, ;
   loRow, ;
   lnJ, ;
   luValue

* Figure out how many columns there are.

lnColumns = toBridge.GetPropertyEx(toTable, 'Columns.Count')

* Store each column name in an array and create a CREATE CURSOR
* statement.

lcCursor = ''
dimension laColumns[lnColumns, 2], laMaxLength[lnColumns]
laMaxLength = 0
for lnI = 0 to lnColumns - 1
   loColumn     = toBridge.GetPropertyEx(toTable, 'Columns[' + ;
      transform(lnI) + ']')
   lcColumnName = loColumn.ColumnName
   lcDataType   = toBridge.GetPropertyEx(loColumn.DataType, 'Name')
   do case
      case inlist(lcDataType, 'Decimal', 'Double', 'Single')
         lcType = 'B(8)'
      case lcDataType = 'Boolean'
         lcType = 'L'
      case lcDataType = 'DateTime'
         lcType = 'T'
      case lcDataType = 'Int'
         lcType = 'I'
      case loColumn.MaxLength = -1
         lcType = 'M'
      otherwise
         lcType = 'C (' + transform(loColumn.MaxLength) + ')'
   endcase
   lcCursor = lcCursor + iif(empty(lcCursor), '', ',') + ;
      lcColumnName + ' ' + lcType + ' null'
   laColumns[lnI + 1, 1] = lcColumnName
   laColumns[lnI + 1, 2] = left(lcType, 1)
   local &lcColumnName
next lnI

* Create the cursor.

lcAlias  = sys(2015)
lcCursor = 'create cursor ' + lcAlias + ' (' + lcCursor + ')'
&lcCursor

* Go through each row, get each column, and populate the cursor.

lnRecords = evl(tnRecords, 9999999999)
lnRows    = toBridge.GetPropertyEx(toTable, 'Rows.Count')
for lnI = 0 to min(lnRows, lnRecords) - 1
   loRow = toBridge.GetPropertyEx(toTable, 'Rows[' + transform(lnI) + ;
         ']')
      for lnJ = 0 to lnColumns - 1
         luValue = toBridge.GetPropertyEx(loRow, 'ItemArray[' + ;
            transform(lnJ) + ']')
         store luValue to (laColumns[lnJ + 1, 1])
         if vartype(luValue) = 'C' or laColumns[lnJ + 1, 2] $ 'CM'
            luValue = transform(luValue)
            laMaxLength[lnJ + 1] = max(laMaxLength[lnJ + 1], len(luValue))
         endif vartype(luValue) = 'C' ...
      next lnJ
      insert into (lcAlias) from memvar
   next lnI

   * Do a final select to get the correct column lengths.

   lcCursor = ''
   for lnI = 1 to alen(laColumns, 1)
      lcColumnName = laColumns[lnI, 1]
      lcType       = laColumns[lnI, 2]
      if lcType $ 'CM' and laMaxLength[lnI] < 255
         lcType = 'C (' + transform(max(laMaxLength[lnI], 1)) + ')'
      endif lcType $ 'CM' ...
      lcCursor = lcCursor + iif(empty(lcCursor), '', ',') + ;
         lcColumnName + ' ' + lcType + ' null'
   next lnI
   lcCursor = 'create cursor ' + tcCursor + ' (' + lcCursor + ')'
   &lcCursor
   append from dbf(lcAlias)
   use in (lcAlias)
   go top
   return
```

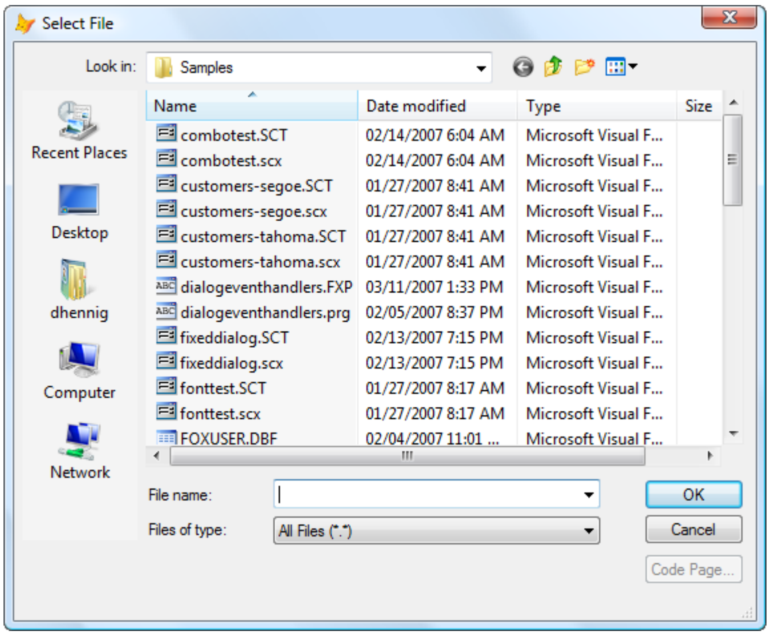
On my machine, this code takes just 1.6 seconds to load the XML into a DataSet and then 14.8 seconds to convert the first DataTable in the DataSet into a VFP cursor, for a total time of 16.4 seconds. That's 60 times faster than using XMLToCursor()! I love taking out the slow parts to make my code faster!

Example #3: file dialogs

VFP developers have relied on GETFILE() and PUTFILE() for years to display file selection dialogs. However, these dialogs have several shortcomings:

- You don't have much control over them: you can specify the file extensions, the title bar caption, and a few other options, but these functions hide most of the settings available in the native Windows dialogs. For example, you can't specify a starting folder or default filename for GETFILE().
- These functions return file names in upper-case rather than the case the user entered or the file actually uses.
- GETFILE() returns only a single filename even though the native dialogs optionally support selecting multiple files.

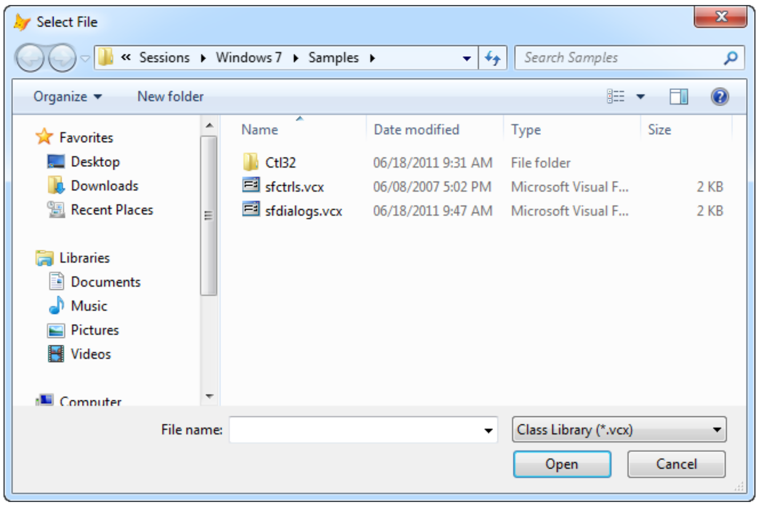
The most important issue, however, is that the dialogs displayed are from the Windows XP era and don't support new features added in Windows Vista and later. **Figure 4** shows the dialog presented by GETFILE(). Although this dialog does have a Recent Places button, it still looks like a dialog from an older operating system.



Name	Date modified	Type	Size
combotest.SCT	02/14/2007 6:04 AM	Microsoft Visual F...	
combotest.scx	02/14/2007 6:04 AM	Microsoft Visual F...	
customers-segoe.SCT	01/27/2007 8:41 AM	Microsoft Visual F...	
customers-segoe.scx	01/27/2007 8:41 AM	Microsoft Visual F...	
customers-tahoma.SCT	01/27/2007 8:41 AM	Microsoft Visual F...	
customers-tahoma.scx	01/27/2007 8:41 AM	Microsoft Visual F...	
dialogeventhandlers.FXP	03/11/2007 1:33 PM	Microsoft Visual F...	
dialogeventhandlers.prg	02/05/2007 8:37 PM	Microsoft Visual F...	
fixeddialog.SCT	02/13/2007 7:15 PM	Microsoft Visual F...	
fixeddialog.scx	02/13/2007 7:15 PM	Microsoft Visual F...	
fonttest.SCT	01/27/2007 8:17 AM	Microsoft Visual F...	
fonttest.scx	01/27/2007 8:17 AM	Microsoft Visual F...	
FOXUSER.DBF	02/04/2007 11:01 ...	Microsoft Visual F...	

Figure 4. The VFP GETFILE() function is an older-looking dialog.

As you can see in **Figure 5**, the Windows 7 open file dialog not only has a more modern interface, it has several features the older dialog doesn't, including back and forward buttons, the "breadcrumb" folder control, and access to Windows Search.



Name	Date modified	Type	Size
Ctl32	06/18/2011 9:31 AM	File folder	
sfctrls.vcx	06/08/2007 5:02 PM	Microsoft Visual F...	2 KB
sfdialogs.vcx	06/18/2011 9:47 AM	Microsoft Visual F...	2 KB

Figure 5. The Windows 7 open file dialog looks modern and has features the older dialog doesn't.

Again, wwDotNetBridge to the rescue. The Dialogs class in Samples.CS, shown in **Listing 7**, has a ShowOpenDialog method that sets the properties of the native file dialog based on properties of the class you can set, such as InitialDir and MultiSelect, displays the dialog, and returns the path of the selected file (multiple files are separated with carriage returns) or blank if the user clicked Cancel.

Listing 7. The Dialogs class provides modern, native file dialogs.

```
public class Dialogs
{
  public string DefaultExt { get; set; }
  public string FileName { get; set; }
  public string InitialDir { get; set; }
  public string Title { get; set; }
  public string Filter { get; set; }
  public int FilterIndex { get; set; }
  public bool MultiSelect { get; set; }

  public string ShowOpenDialog()
  {
    string fileName = "";
```

```
      OpenFileDialog dialog = new OpenFileDialog();
      dialog.FileName = FileName;
      dialog.DefaultExt = DefaultExt;
      dialog.InitialDirectory = InitialDir;
      dialog.Title = Title;
      dialog.Filter = Filter;
      dialog.FilterIndex = FilterIndex;
      dialog.Multiselect = MultiSelect;

      if (dialog.ShowDialog() == DialogResult.OK)
      {
      if (dialog.FileNames.Length > 0)
      {
        foreach (string file in dialog.FileNames)
        {
          fileName += file + "\n";
        }
      }
        else
        {
          fileName = dialog.FileName;
        }
      }
      else
      {
        fileName = "";
      }
      return fileName;
    }
  }
```

Listing 8 shows some VFP code, taken from TestOpenFile.PRG, which uses the Dialogs class. It sets the initial folder to the FFC\Graphics folder in the VFP home directory and turns on MultiSelect so you can select multiple files.

Listing 8. TestOpenFile.prg uses Dialogs to display a file dialog.

```
local loBridge, ;
  loFox, ;
  lcFile

* Create the wwDotNetBridge object.

loBridge = newobject('wwDotNetBridge', 'wwDotNetBridge.prg', '', 'V4')

* Load our assembly and instantiate the Dialogs class.

loBridge.LoadAssembly('Samples.dll')
loFox = loBridge.CreateInstance('Samples.Dialogs')

* Set the necessary properties, then display the dialog.

loFox.FileName    = 'add.bmp'
loFox.InitialDir  = home() + 'FFC\Graphics'
loFox.Filter      = 'Image Files (*.bmp, *.jpg, *.gif)|*.bmp;*.jpg;' + ;
    '*.gif|All files (*.*)|*.*'
loFox.Title       = 'Select Image'
loFox.MultiSelect = .T.
lcFile            = loFox.ShowOpenDialog()
if not empty(lcFile)
  messagebox('You selected ' + lcFile)
endif not empty(lcFile)
```

The dialog displayed when you run this program is based on the operating system. Under Windows XP, it'll look like an XP dialog. Under Windows 8, it'll look like a Windows 8 dialog.

Example #4: writing to the Windows Event Log

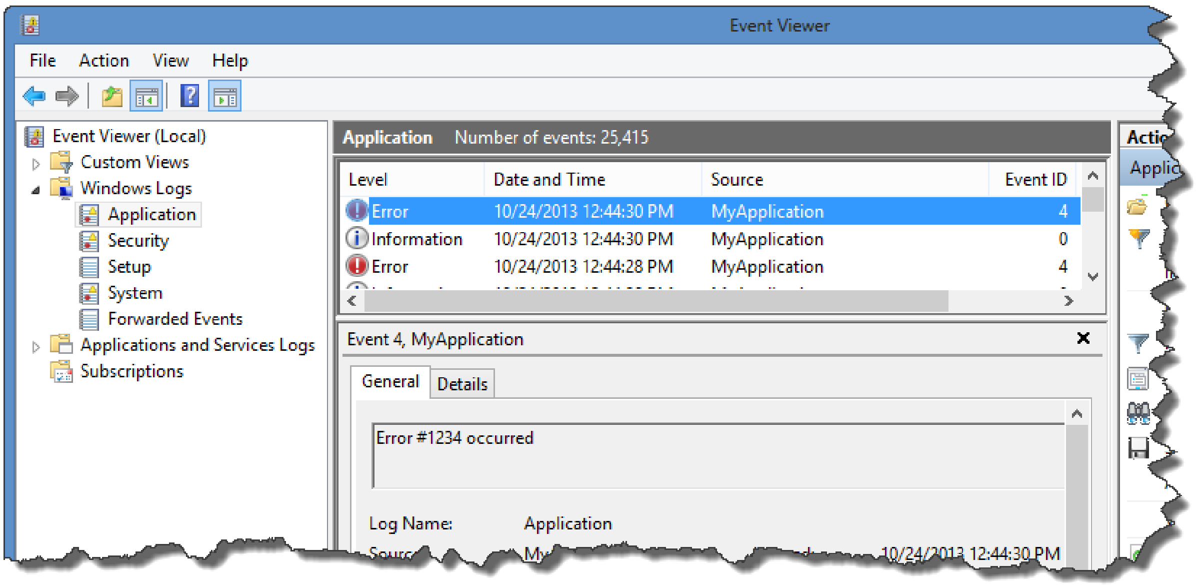
Although I prefer to log diagnostic information and errors to text and/or DBF files, some people like to log to the Windows Event Log, as system administrators are used to looking there for issues. Doing that from a VFP application is ugly because the Win32 API calls are messy. Using .NET via wwDotNetBridge, on the other hand, is very easy. The code in **Listing 9**, taken from WindowsEventLog.prg that accompanies this document, shows how to do it in just a few lines of code.

Listing 9. wwDotNetBridge makes it easy to write to the Windows Event Log.

```
local loBridge, ;
   lcSource, ;
   lcLogType, ;
   lcClass, ;
   loValue

* Create the wwDotNetBridge object.

loBridge = newobject('wwDotNetBridge', 'wwDotNetBridge.prg', '', 'V4')

* Define the source and log types.

lcSource  = 'MyApplication'
lcLogType = 'Application'

* Put the name of the .NET class we'll use into a variable so we don't
* have to type it on every method call.

lcClass = 'System.Diagnostics.EventLog'

* See if the source already exists; create it if not.

if not loBridge.InvokeStaticMethod(lcClass, 'SourceExists', lcSource)
   loBridge.InvokeStaticMethod(lcClass, 'CreateEventSource', lcSource, ;
      lcLogType)
endif not loBridge.InvokeStaticMethod ...

* Create an information message.

loBridge.InvokeStaticMethod(lcClass, 'WriteEntry', lcSource, ;
   'Some application event that I want to log')

* For an error message, we need to use an enum. Normally we'd use this:

*loValue = ;
*  loBridge.GetEnumValue('System.Diagnostics.EventLogEntryType.Error')

* However, that doesn't work in this case due to the way WriteEntry
* works, so we'll use this method instead.

loValue = loBridge.CreateComValue()
loValue.SetEnum('System.Diagnostics.EventLogEntryType.Error')
loBridge.InvokeStaticMethod(lcClass, 'WriteEntry', lcSource, ;
   'Error #1234 occurred', loValue, 4)

* Display the last 10 logged events.

loEventLog        = loBridge.CreateInstance(lcClass)
loEventLog.Source = lcSource
loEventLog.Log    = lcLogType
loEvents          = loBridge.GetProperty(loEventLog, 'Entries')
lcEvents          = ''
for lnI = loEvents.Count - 1 to loEvents.Count - 10 step -1
   loEvent  = loEvents.Item(lnI)
```

```
    lcEvents = lcEvents + transform(lnI) + ': ' + loEvent.Message + ;
      chr(13)
next lnI
messagebox('There are ' + transform(loEvents.Count) + ' events:' + ;
    chr(13) + chr(13) + lcEvents)
```

This code calls static methods of the .NET System.Diagnostics.EventLog class. It starts by checking whether the event source, which is usually an application name or something equally descriptive, already exists or not; if not, it's created using the specified type (Application, Security, or System; Application in this case). It then calls the WriteEntry method to write to the log. The first call to WriteEntry uses one of the overloads of that method: the one expecting the name of the source and the message. The second call uses a different overload: the one expecting the name of the source, the message, an enum representing the type of entry, and a user-defined event ID (4 in this case). Note the comment about how enums normally work but an alternative method that's needed in this case. **Figure 6** shows how the log entries appear in the Windows Event Viewer.



Application Number of events: 25,415

Level	Date and Time	Source	Event ID
Error	10/24/2013 12:44:30 PM	MyApplication	4
Information	10/24/2013 12:44:30 PM	MyApplication	0
Error	10/24/2013 12:44:28 PM	MyApplication	4

Event 4, MyApplication

Error #1234 occurred

Log Name: Application

Figure 6. The results of running WindowsEventLog.prg.

To display the log entries, the code creates an instance of System.Diagnostics.EventLog and goes through the Entries collection.

Example #5: starting and stopping processes

The VFP RUN command allows you to run an external process but you don't have much control over it. For example, it only supports running an EXE, so you can't specify the name of a Microsoft Word document to have it open that document; you have to know the location of Word.exe and pass it the name of the document on the command line. Lots of developers like to use the Win32 API ShellExecute function because it does allow you to specify the name of a file to open that file in whatever application it's associated with. However, again you don't have a lot of control, such as the ability to kill the application once you're done with it.

.NET's System.Diagnostics.Process class makes it easy to do as the code in **Listing 10** illustrates.

Listing 10. .NET makes it easy to start and stop a process.

```
* Open a text file in Notepad.

loBridge = newobject('wwDotNetBridge', 'wwDotNetBridge.prg', '', 'V4')
loProcess = loBridge.CreateInstance('System.Diagnostics.Process')
loBridge.InvokeMethod(loProcess, 'Start', 'text.txt')

* Let the user tell use when to proceed.

messagebox("Now we'll kill Notepad")

* Find the process ID.
```

```
loProcesses = ;
   loBridge.InvokeStaticMethod('System.Diagnostics.Process', ;
   'GetProcesses')
lnID       = -1
for lnI = 0 to loProcesses.Count - 1
   loProcess = loProcesses.Item(lnI)
   if loProcess.ProcessName = 'notepad'
      lnID = loProcess.ID
      exit
   endif loProcess.ProcessName = 'notepad'
next lnI

* If we found it, kill it.

if lnID > -1
   loProcess = ;
      loBridge.InvokeStaticMethod('System.Diagnostics.Process', ;
      'GetProcessById', lnID)
   loProcess.Kill()
endif lnID > -1

* We can also do it this way, which has the advantage that we know the
* process ID without having to look for it.

loProcess = loBridge.InvokeStaticMethod(lcClass, 'Start', 'text.txt')
messagebox("Now we'll kill Notepad")
loProcess.Kill()
```

Example #6: how I use wwDotNetBridge

I use wwDotNetBridge extensively in an application called Stonefield Query Studio.NET. Studio.NET is a VFP application that manages a data dictionary and other configuration files for Stonefield Query.NET, a new .NET version of our Stonefield Query product. (Studio.NET is currently a VFP application rather than a .NET application because it was easier to convert our existing Studio application than to write one from scratch in .NET.) We decided to use properties and methods of .NET classes to read from and write to the configuration files rather than doing it directly in VFP for a variety of reasons:

- The data dictionary is stored in a SQLite database. Although SQLite can be used from VFP, it requires installing an ODBC driver to do so. .NET doesn't use ODBC to access SQLite; it uses an assembly to talk directly to the database instead. So, by using the .NET class we already have to access the database, that's one less thing to install on the user's system.
- Some of the access logic uses business rules. For example, some information is encrypted so the encryption and decryption login already in the .NET classes would have to be duplicated in VFP. Another example: when a table is deleted, the .NET data dictionary code automatically deletes all fields and joins belonging to that table. That code too would have to be duplicated in VFP if the data dictionary was accessed directly. By using the existing .NET code, it's a simple matter of calling the appropriate method.
- Partway through development, we changed how the data dictionary is stored; we originally stored it in a set of XML files. If we'd accessed the data dictionary directly in VFP, we'd have to rewrite all of the access code twice: once in .NET and once in VFP. By using .NET methods for the access, we only had to rewrite the .NET portion; the method calls in the VFP code stayed the same.

Our code makes extensive use of wwDotNetBridge's GetPropertyEx, SetPropertyEx, and InvokeMethod to access properties and methods. In some cases, we added helper methods to the .NET code to work better in VFP. For example, there's a table collection in the .NET data dictionary. Normally, a .NET client can use LINQ to get any set of tables it wants, such as just tables in a specific database or only those that the user can see ("reportable" tables). However, VFP can't use LINQ so we created a GetItems method that accepts a set of parameters to filter the collection and returns an array of the tables matching the filter. **Listing 11** shows the code for GetItems.

Listing 11. The .NET GetItems method returns an array of just those tables matching the specified filter conditions.

```
public ITable[] GetItems(string database, bool reportableOnly,
   string filter, string updated)
{
   bool checkFilter = !String.IsNullOrWhiteSpace(filter);
   string upperFilter = checkFilter ? filter.ToUpper() : "";
   DateTime testDate = String.IsNullOrWhiteSpace(updated) ?
      DateTime.MinValue : DateTime.Parse(updated);
   bool checkDB = !String.IsNullOrWhiteSpace(database);
   var tables = this.Where(t => (!checkFilter || t.Name.ToUpper()
      .Contains(upperFilter)
      || t.Caption.ToUpper().Contains(upperFilter))
      && (!reportableOnly || t.Reportable)
      && t.Updated >= testDate
      && (!checkDB || t.Database.ID.ToString() == database))
      .ToArray<ITable>();
   return tables;
}
```

Listing 12 shows part of the VFP code that calls the GetItems method.

Listing 12. Part of the code used to populate an array with desired tables.

```
lparameters taArray, ;
   tcDatabase, ;
   tlReportableOnly, ;
   tcFilter, ;
   tdUpdated
loItems = oBridge.InvokeMethod(This.oMetaData, 'GetItems', ;
   evl(tcDatabase, ''), tlReportableOnly, evl(tcFilter, ''), ;
   evl(tdUpdated, ''))
```

Although the .NET data dictionary objects could be accessed directly in VFP, doing so would be cumbersome because some of the properties in those objects aren't accessible in VFP. For example, every object has an ID property that's a Guid, and as we saw earlier, Guids aren't available in VFP. So, we created VFP wrapper classes for the .NET classes and the GetObject and SaveItem methods take care of reading and writing the properties to and from the .NET and VFP objects. For example, **Listing 13** shows some of the code in the GetObject method to convert the properties of a .NET Table object to a VFP wrapper object. Notice how Guids are handled: the GuidString property of the object returned by GetPropertyEx is something added by wwDotNetBridge; it contains the Guid value converted to a string.

Listing 13. The VFP GetObject method reads .NET properties and writes them to a VFP wrapper class.

```
lparameters toItem
local loItem, ;
   loID, ;
   loDatabase, ;
   loOriginalTable, ;
   loDataGroups, ;
   lnDataGroups, ;
   lcDataGroups, ;
   lnI, ;
   loDataGroup
loItem = This.Add()
with loItem
   loID            = oBridge.GetPropertyEx(toItem, 'ID')
   .cID            = loID.GuidString
   .cAlias         = oBridge.GetPropertyEx(toItem, 'Name')
   .cCaption       = oBridge.GetPropertyEx(toItem, 'Caption')
   loDatabase      = oBridge.GetPropertyEx(toItem, 'Database')
   loID            = oBridge.GetPropertyEx(loDatabase, 'ID')
   .cDatabase      = loID.GuidString
   .lReportable    = oBridge.GetPropertyEx(toItem, 'Reportable')
   loOriginalTable = oBridge.GetPropertyEx(toItem, 'OriginalTable')
   if vartype(loOriginalTable) = 'O'
```

```
      loID            = oBridge.GetPropertyEx(loOriginalTable, 'ID')
      .cOriginalTable = loID.GuidString
   endif vartype(loOriginalTable) = 'O'

* Handle datagroups.

   loDataGroups = oBridge.GetPropertyEx(toItem, 'DataGroups')
   lnDataGroups = oBridge.GetPropertyEx(loDataGroups, 'Count')
   lcDataGroups = ''
   for lnI = 1 to lnDataGroups
      loDataGroup  = oBridge.GetPropertyEx(toItem, ;
         'DataGroups[' + transform(lnI - 1) + ']')
      loID         = oBridge.GetPropertyEx(loDataGroup, 'ID')
      lcDataGroups = lcDataGroups + loID.GuidString + ccCRLF
   next lnI
   .cDataGroup = lcDataGroups
endwith
return loItem
```

Saving changes to an item is better handled in .NET, especially since a new item needs to be assigned a new Guid, so the SaveItemStudio method (**Listing 14**) takes care of that. It accepts the VFP wrapper object as a dynamic, meaning it doesn't do compile-time data type checking. Notice how it handles a new object: if the cID property isn't filled in, a new object is created. Otherwise, the object with the specified ID is retrieved from the collection.

Listing 14. The SaveItemStudio method accepts a VFP wrapper classes and writes to a .NET object.

```
public override ITable SaveItemStudio(dynamic item)
{
   Table saveItem;
   string name = item.cAlias;
   if (String.IsNullOrWhiteSpace(item.cID))
   {
      saveItem = (Table)New(name);
   }
   else
   {
      Guid id = Guid.Parse(item.cID);
      saveItem = (Table)this[id];
   }
   saveItem.Caption = item.cCaption;
   saveItem.databaseGUID = item.cDatabase;
   saveItem.OriginalTable = null;
   saveItem.originalTableID = item.cOriginalTable;
   saveItem.Reportable = item.lReportable;
   DeserializeDataGroups(saveItem, item.cDataGroup);
   SaveItem(saveItem);
   return saveItem;
}
```

Because VFP can't access .NET generics, it can't create a List<string>, but some of the methods Studio.NET calls require one as a parameter. So, a GetList helper method takes care of that:

```
public List<string> GetList()
{
   return new List<string>();
}
```

The VFP code calls GetList to create a list, then uses oBridge.InvokeMethod(loList, 'Add', 'SomeValue') to add a value to the list. For example, to browse the contents of a table, the VFP BrowseTable method calls GetList to create a list, adds the name of the fields to the list, calls the RetrieveDataForFields method of the .NET DataEngine object to retrieve a DataTable containing the results, then calls the same CreateCursorFromDataTable function we saw earlier to convert the DataTable to a VFP cursor so it can be browsed.

References

Here are links to articles and documentation about wwDotNetBridge and .NET interop:

- wwDotNetBridge home page: http://www.west-wind.com/wwDotnetBridge.aspx
- wwDotNetBridge documentation: http://tinyurl.com/ltagjhk
- wwDotNetBridge white paper: http://tinyurl.com/lclaflx
- “Using .NET Components via COM from Visual FoxPro (or other COM client)”: http://tinyurl.com/ycn4bl

Summary

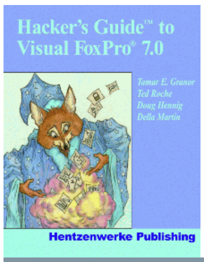
wwDotNetBridge makes it easy to call .NET code from VFP. It eliminates the need to add special directives to the .NET code so it can be used with COM and the need to register the component on the user’s system. It also takes care of the differences between .NET and VFP in dealing with arrays and other data types. This means you can create small .NET classes that accomplish tasks difficult to do or that run slowly in VFP and easily call them in your applications to add new capabilities or speed up processing. Download wwDotNetBridge and try it out for yourself.

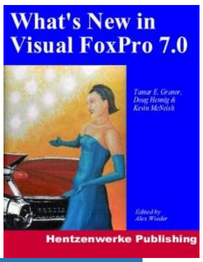
Biography

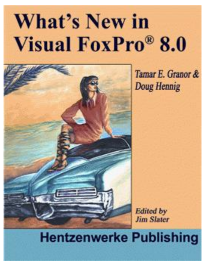
Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

Doug is co-author of *VFPX: Open Source Treasure for the VFP Developer*, *Making Sense of Sedna and SP2*, the *What’s New in Visual FoxPro* series, *Visual FoxPro Best Practices For The Next Ten Years*, and *The Hacker’s Guide to Visual FoxPro 7.0*. He was the technical editor of *The Hacker’s Guide to Visual FoxPro 6.0* and *The Fundamentals*. All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (http://www.foxrockx.com).

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox and Southwest Xbase++ conferences (http://www.swfox.net). He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.codeplex.com). He was a Microsoft Most Valuable Professional (MVP) from 1996 through 2011. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).






What's New in Visual FoxPro® 8.0
Tamar E. Granor & Doug Hennig
Hentzenwerke Publishing