Session E-WDNB

Practical Uses of wwDotNetBridge to Extend Your VFP Applications

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

Overview

wwDotNetBridge lets you call just about any .NET code directly from VFP and helps overcome most of the limitations of regular .NET COM interop. This library by Rick Strahl allows you to provide .NET functionality to your VFP applications that wouldn't otherwise be available. In this session, you'll see many practical examples that show how you can add new capabilities to your applications that would be difficult or impossible to achieve natively from VFP.

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 versions 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 these functions via .NET is easier anyway.

Fortunately, there are various mechanisms that allow you to access .NET code from VFP applications. For example, my "Creating ActiveX Controls for VFP Using .NET" white paper, available at http://doughennig.com/papers/default.html, shows 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.

Rick Strahl created 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. I believe that wwDotNetBridge is the future of VFP.

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 1**.

There are other causes for the "unable to load Clr instance" error as well: see http://tinyurl.com/yabovc3k for several solutions.

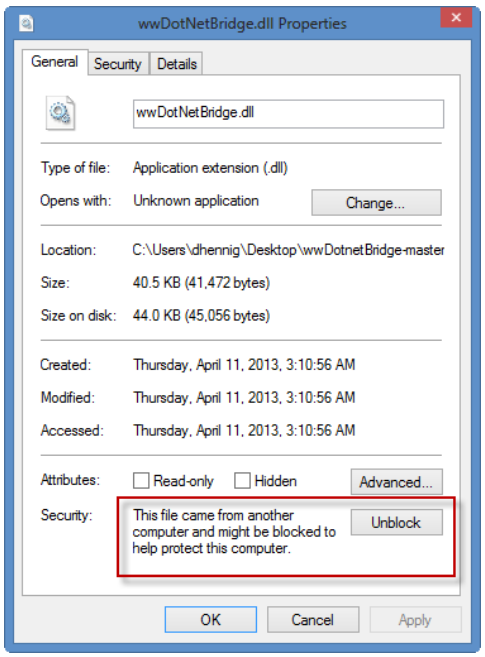

Figure 1. Unblock wwDotNetBridge.DLL to prevent errors when using it.

Using wwDotNetBridge

Start by instantiating the wwDotNetBridge wrapper class using code like:

```
loBridge = newobject('wwDotNetBridge', 'wwDotNetBridge.prg', '', 'V2')
```

The last parameter tells wwDotNetBridge which version of the .NET runtime to load. By default, it loads version 4.0; this example specifies version 2.0. 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 easiest way to do that is:

```
do wwDotNetBridge
loBridge = GetwwDotNetBridge()
```

GetwwDotNetBridge uses a public variable named __DOTNETBRIDGE to cache the instantiated wwDotNetBridge object, only instantiating it if the variable doesn't exist or doesn't contain an object.

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 SMTPLibrary assembly we'll see later and instantiates the SMTP class which lives in the SMTPLibrary namespace:

```
loBridge.LoadAssembly('SMTPLibrary.dll')
loMail = loBridge.CreateInstance('SMTPLibrary.SMTP')
```

Note that you need to specify the correct path for the DLL (for example, "Samples\C#\SMTPLibrary\SMTPLibrary\bin\Debug\SMTPLibrary.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 sets some properties and calls a method:

```
loMail.MailServer = 'smtp.gmail.com'
loMail.Username   = 'doug.o.hennig@gmail.com'
loMail.SendMail()
```

Some types of properties, such as properties that are collections or lists, can't be accessed directly. For those, use the GetPropertyEx method of wwDotNetBridge to get the property's value or SetPropertyEx to set it:

```
loTable = loBridge.GetPropertyEx(loDS, 'Tables[0]')
```

For a static property, use GetStaticProperty and SetStaticProperty.

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

```
lcResult = loBridge.InvokeMethod(tuValue, 'ToString')
loProcesses = loBridge.InvokeStaticMethod('System.Diagnostics.Process', ;
   'GetProcesses')
```

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. Include this PRG in your project so it's built into the EXE.
- wwDotNetBridge.dll: a .NET DLL that handles all of the interop stuff. Distribute this file with your application.
- ClrHost.dll: a custom version of the .NET runtime host. Distribute this file with your application.

The architecture of wwDotNetBridge is shown in **Figure 2**, an updated version of the diagram that appears in Rick's documentation.

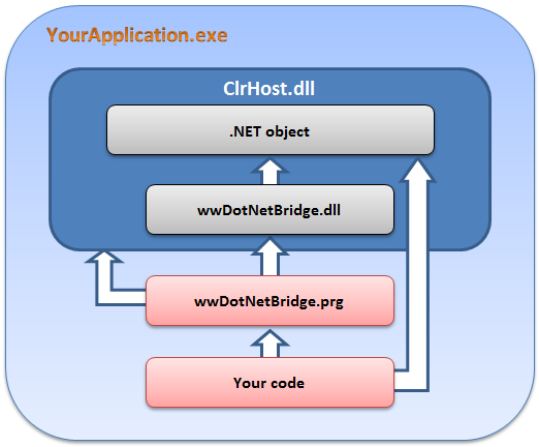

Figure 2. The architecture of wwDotNetBridge.

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.

Debugging .NET code from VFP

Debugging COM objects is hard. Typically, you'll do it the old fashioned way: add statements to display the current value of variables or properties, run the code, see what the values are, fix the code, run it again, … and so on.

Debugging .NET code called from VFP via wwDotNetBridge is easy (assuming you have the source code):

- Open the .NET solution in Visual Studio.
- Set a breakpoint where desired.
- Start VFP.
- Choose Attach to Process from the Visual Studio Debug menu and choose VFP.EXE.
- Execute the VFP code that calls the desired .NET method.

When the breakpoint in the .NET code is hit, the Visual Studio debugger kicks in. You can do all the usual debugging things: step through the code, examine and change the values of properties and variables, skip over code, etc. When you're finished debugging, choose Detach All from the Debug menu.

Note that if you typically copy the .NET DLL from the bin folder where it's created into the application folder, you also need to copy the associated PDB (debugging information) file into the folder or debugging won't work.

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

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 1**, taken from WindowsEventLog.prg that accompanies this document, shows how to do it in just a few lines of code.

Listing 1. wwDotNetBridge makes it easy to write to the Windows Event Log.

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

* Create the wwDotNetBridge object.

do wwDotNetBridge
loBridge = GetwwDotNetBridge()

* 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 3** shows how the log entries appear in the Windows Event Viewer.

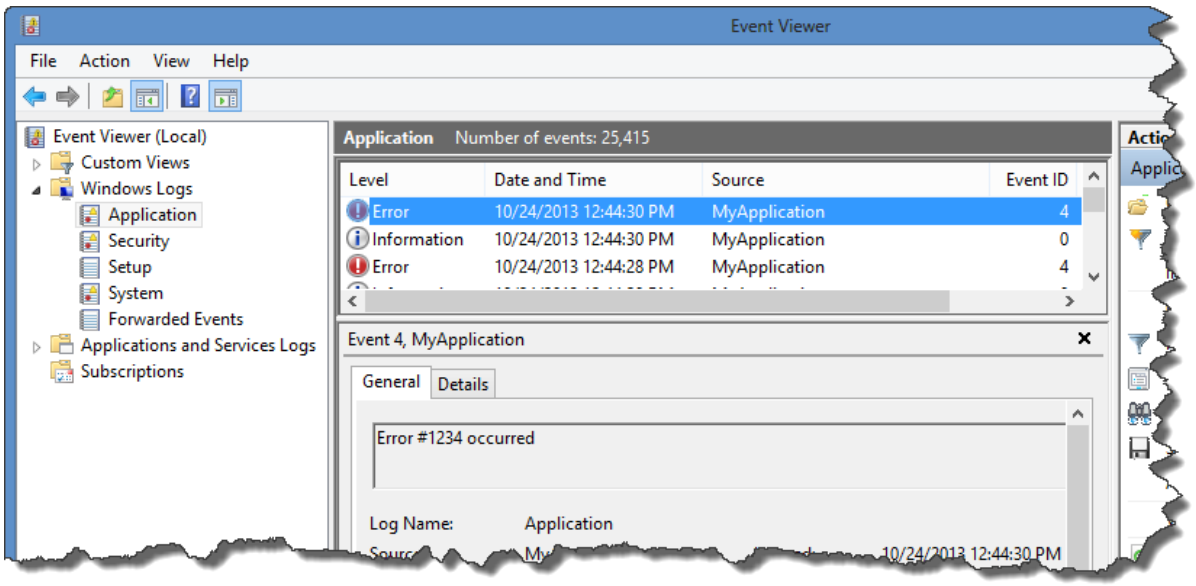

Figure 3. The results of running WindowsEventLog.prg.

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

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 just 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 2**, taken from Processes.prg, illustrates.

Listing 2. .NET makes it easy to start and stop a process.

```
* Open a text file in Notepad.

do wwDotNetBridge
loBridge  = GetwwDotNetBridge()
```

```
lcClass   = 'System.Diagnostics.Process'
loProcess = loBridge.CreateInstance(lcClass)
loBridge.InvokeMethod(loProcess, 'Start', 'text.txt')

* Find the process ID.

loProcesses = loBridge.InvokeStaticMethod(lcClass, 'GetProcesses')
lnID        = -1
for lnI = 0 to loProcesses.Count - 1
   loProcess = loProcesses.Item(lnI)
   if loProcess.ProcessName = 'notepad'
      lnID = loProcess.ID
* We could kill it here but we'll do it a different way below.
*     loProcess.Kill()
      exit
   endif loProcess.ProcessName = 'notepad'
next lnI

* If we found it, kill it.

if lnID > -1
   messagebox("Now we'll kill Notepad")
   loProcess = loBridge.InvokeStaticMethod(lcClass, '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("Here's another instance that we'll also kill")
loProcess.Kill()
```

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 3 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 3. 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 4 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 create a VFP cursor from the first DataTable in the DataSet.

Listing 4. 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.

do wwDotNetBridge
loBridge = GetwwDotNetBridge()

* 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)
local lnColumns, ;
   lcNull, ;
   lcCursor, ;
   laColumns[1], ;
   laMaxLength[1], ;
   lnI, ;
   loColumn, ;
   lcColumnName, ;
   lcDataType, ;
   lcType, ;
   lcAlias, ;
   lnRecords, ;
   lnRows, ;
   loRow, ;
   lnJ, ;
   luValue, ;
   lcError, ;
   laFields[1], ;
   lnFields, ;
   lnV, ;
   lcVrbl, ;
   lcFType, ;
   llSame, ;
   llError

* Figure out how many columns there are.

lnColumns = min(toBridge.GetPropertyEx(toTable, 'Columns.Count'), 254)
if lnColumns = 0
   return .F.
endif lnColumns = 0

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

lcNull = set('NULL')
set null on
lcCursor = ''
dimension laColumns[lnColumns, 3], laMaxLength[lnColumns]
laMaxLength = 0
for lnI = 0 to lnColumns - 1
```

```
      loColumn     = toBridge.GetPropertyEx(toTable, 'Columns[' + ;
         transform(lnI) + ']')
      lcColumnName = GetValidName(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 = 'Byte[]'
            lcType = 'W'
         case inlist(lcDataType, 'Int', 'Byte')
            lcType = 'I'
         case lcDataType = 'Guid'
            lcType = 'C(36)'
         case loColumn.MaxLength = -1
            lcType = 'M'
         otherwise
            lcType = 'C(' + transform(loColumn.MaxLength) + ')'
      endcase
      if inlist(upper(lcColumnName) + ' ', 'NULL ', 'NOT ', 'CHECK ', 'ERROR ', ;
         'AUTOINC ', 'NEXTVALUE ', 'STEP ', 'DEFAULT ', 'PRIMARY ', 'KEY ', ;
         'UNIQUE ', 'COLLATE ', 'REFERENCES ', 'TAG ', 'NOCPTRANS ')
         lcColumnName = '[' + lcColumnName + ']'
      endif inlist(upper(lcColumnName) ...
      lcCursor = lcCursor + iif(empty(lcCursor), '', ',') + lcColumnName + ;
         ' ' + lcType
      laColumns[lnI + 1, 1] = lcColumnName
      laColumns[lnI + 1, 2] = left(lcType, 1)
      laColumns[lnI + 1, 3] = lcDataType
      local &lcColumnName
      if len(lcCursor) > 8000
         exit
      endif len(lcCursor) > 8000
   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) + ']')
         do case
            case inlist(laColumns[lnJ + 1, 3], 'Object', 'Type')
               luValue = toBridge.InvokeMethod(luValue, 'ToString')
               laMaxLength[lnJ + 1] = max(laMaxLength[lnJ + 1], len(luValue))
            case laColumns[lnJ + 1, 3] = 'Guid'
               luValue = luValue.GuidString
               laMaxLength[lnJ + 1] = 36
            case vartype(luValue) = 'C' or laColumns[lnJ + 1, 2] $ 'CM'
               luValue = transform(luValue)
```

```
            laMaxLength[lnJ + 1] = max(laMaxLength[lnJ + 1], len(luValue))
      endcase
      store luValue to (laColumns[lnJ + 1, 1])
   next lnJ
   try
      insert into (lcAlias) from memvar
   catch

* If we have a data type issue, indicate which column.

      lcError  = ''
      lnFields = afields(laFields)
      for lnV = 1 to lnFields
         lcVrbl  = 'M.' + laFields[lnV, 1]
         lcFType = laFields[lnV, 2]
         lcType  = type(lcVrbl)
         llSame  = lcType = lcFType or ;
            (lcType = 'N' and lcFType $ 'NFIBY') or ;
            (lcType $ 'CMV' and lcFType $ 'CMV') or ;
            empty(lcType)
         if not llSame
            lcError = lcError + lcVrbl + ' Variable type: ' + lcType + ;
               ' Field type: ' + lcFType + chr(13)
         endif not llSame
      next lnV
      llError = .T.
   endtry
   if llError
      exit
   endif llError
next lnI

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

if not llError
   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
      if len(lcCursor) > 8000
         exit
      endif len(lcCursor) > 8000
   next lnI
   lcCursor = 'create cursor ' + tcCursor + ' (' + lcCursor + ')'
   &lcCursor
   append from dbf(lcAlias)
   go top
endif not llError
if lcNull = 'OFF'
   set null off
endif lcNull = 'OFF'
use in (lcAlias)
if llError
   error lcError
endif llError
return not llError
```

```
function GetValidName(tcName)
local lcIllegal, ;
   lcDelimiters, ;
   lcName

* Decide what are illegal characters and the usual suspects for field
* delimiters.

lcIllegal    = [ ~!@#$%^&*()-+=-/?{}|<>,:;\.']
lcDelimiters = '"[]`'

* Convert illegal characters to _ and strip out delimiters.

lcName = chrtran(lcName, lcIllegal, replicate('_', len(lcIllegal)))
lcName = chrtran(lcName, lcDelimiters, '')
lcName = iif(isdigit(left(lcName, 1)), 'A', '') + lcName
return lcName
```

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!

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 uppercase 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 available in more modern versions. **Figure 4** shows the dialog presented by GETFILE(). Although this dialog does have quick access buttons at the left, it still looks like a dialog from an older operating system.

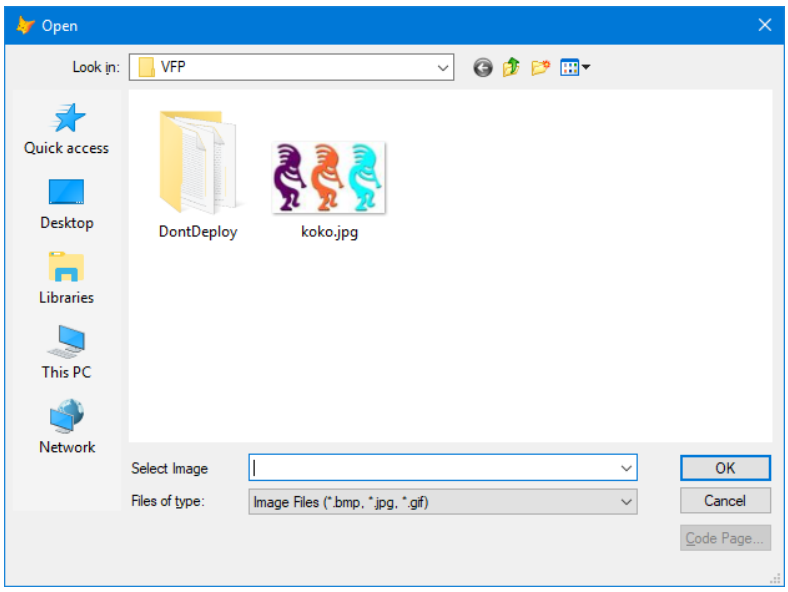

Figure 4. The VFP GETFILE() function is an older-looking dialog.

As you can see in **Figure 5**, the Windows 10 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.

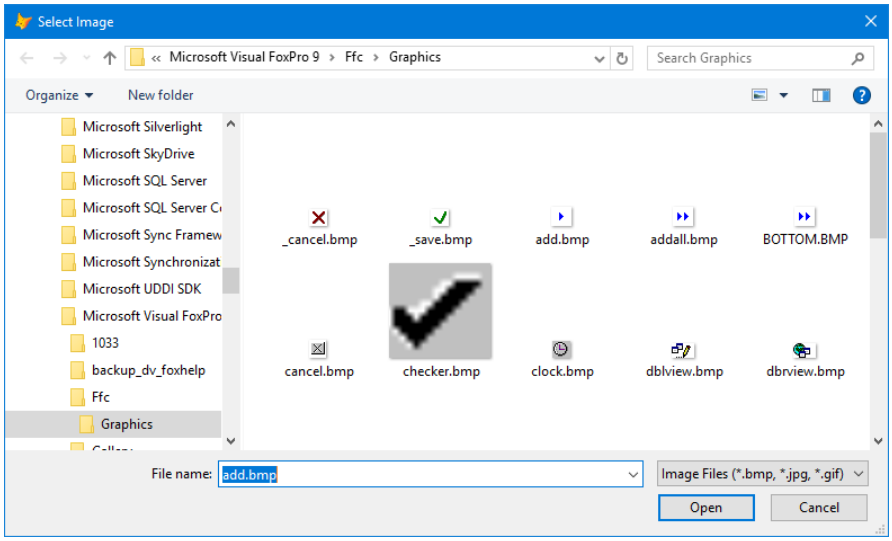

Figure 5. The Windows 10 open file dialog looks modern and has features the older dialog doesn't.

Again, wwDotNetBridge to the rescue. The Dialogs class in Dialogs.cs, shown in **Listing 5**, 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 5. 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 6 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 6. TestOpenFile.prg uses Dialogs to display a file dialog.

```
local loBridge, ;
  loFox, ;
  lcFile

* Create the wwDotNetBridge object.

do wwDotNetBridge
loBridge = GetwwDotNetBridge()

* Load our assembly and instantiate the Dialogs class.

loBridge.LoadAssembly('FileDialog.dll')
loFox = loBridge.CreateInstance('FileDialog.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 10, it'll look like a Windows 10 dialog.

Encryption

Many applications need encryption to store things like connection strings, user names and passwords, and so on. I've used Craig Boyd's VFPEncryption library for many years with great success. However, recently we had a U.S. federal government department as a potential customer who insisted that we use FIPS-compliant encryption or they wouldn't legally be able to use our software. FIPS (Federal Information Processing Standards) 140 (https://en.wikipedia.org/wiki/FIPS_140-2) has numerous requirements, all of which we were able to implement except one: VFPEncryption.fll isn't certified for FIPS.

It turns out several .NET encryption classes are FIPS 140-certified, including AesCryptoServiceProvider. So, it was a simple matter (thanks to Stack Overflow, http://tinyurl.com/ybjvs9cy) to create a small .NET DLL that uses AesCryptoServiceProvider to encrypt and decrypt strings. **Listing 7** shows a partial listing (most of the code is omitted; see Encryption.cs in the samples accompanying this document for details).

Listing 7. Encryption.cs (without the code details) provides FPS 140-compliant encryption.

```
using System;
using System.Text;
using System.Security.Cryptography;

namespace Encryption
{
    public class Encryption
    {
        public static string Encrypt(string toEncrypt, string key, string IV)
        {
            AesCryptoServiceProvider provider = new AesCryptoServiceProvider();
            // Rest of code omitted for brevity
        }

        public static string Decrypt(string toDecrypt, string key, string IV)
        {
            AesCryptoServiceProvider provider = new AesCryptoServiceProvider();
            // Rest of code omitted for brevity
        }
    }
}
```

The Encrypt and Decrypt methods expect three parameters:

- The string to encrypt or decrypt.

- The encryption key.
- The initialization vector (IV); see http://tinyurl.com/cp3tpsw for a description.

Because these are static methods, they're easy to call from VFP using wwDotNetBridge using InvokeStaticMethod.

One problem with switching encryption mechanisms is what do you do with existing customers? Their existing encrypted strings can't be correctly decrypted using the new encryption library. So, I decided to add a flag character, CHR(1), to the end of strings encrypted with the new mechanism. If that character is there, use the new library to decrypt the string. If not, use the old.

EncryptString.prg (**Listing 8**) encrypts a string using the .NET Encryption.dll. We don't worry about whether or not to use VFPEncryption.fll here; all new encryption is done using the new library.

Listing 8. EncryptString.prg uses Encryption.dll to encrypt a string.

```
lparameters tcString, ;
   tcKey

* Handle a blank string or key.

if empty(tcString) or empty(tcKey)
   return tcString
endif empty(tcString) ...

* The new code uses Encryption.DLL to do the encrypting. Add a CHR(1) to the
* end so Decrypt can tell whether a string was encrypted with new or old code.

return oBridge.InvokeStaticMethod('Encryption.Encryption', 'Encrypt', ;
   trim(tcString), tcKey, chr(102) + chr(57) + chr(110) + chr(73) + ;
   chr(73) + chr(71) + chr(97) + chr(89) + chr(70) + chr(51) + chr(54) + ;
   chr(97) + chr(69) + chr(55) + chr(48) + chr(68)) + chr(1)
```

DecryptString.prg (**Listing 9**), on the other hand, has to determine which mechanism was used to encrypt a string and use the same one to decrypt it. If the right-most character is CHR(1), it strips that off and uses Encryption.dll; otherwise, it uses VFPEncryption.fll.

Listing 9. DecryptString.prg uses either VFPEncryption.fll or Encryption.dll to decrypt a string.

```
lparameters tcString, ;
   tcKey
local lcString

* Handle a blank string or key.

if empty(tcString) or empty(tcKey)
   return tcString
endif empty(tcString) ...

* If the right-most character of the encrypted string is CHR(1), the string
* was encrypted using Encryption.dll, so use that to decrypt it.

lcString = trim(tcString)
if right(lcString, 1) = chr(1)
   lcString = left(lcString, len(lcString) - 1)
```

```
   return nvl(oBridge.InvokeStaticMethod('Encryption.Encryption', 'Decrypt', ;
      lcString, tcKey, chr(102) + chr(57) + chr(110) + chr(73) + chr(73) + ;
      chr(71) + chr(97) + chr(89) + chr(70) + chr(51) + chr(54) + chr(97) + ;
      chr(69) + chr(55) + chr(48) + chr(68)), '')

* The string was encrypted using VFPEncryption.fll, so use that to decrypt it.

else
   if not 'vfpencryption.fll' $ lower(set('LIBRARY'))
      lcLibrary = 'VFPEncryption.fll'
      set library to (lcLibrary) additive
   endif not 'vfpencryption.fll' $ lower(set('LIBRARY'))
   return Decrypt(lcString, tcKey, 1024)
endif right(lcString, 1) = chr(1)
```

One thing to note about both EncryptString.prg and DecryptString.prg is that neither stores the initialization vector in a memory variable. Instead, it's passed directly to the method as a series of bytes. The same is true for the decrypted result in DecryptString.prg. This helps minimize hacking.

Both EncryptString.prg and DecryptString.prg expect that oBridge contains a reference to wwDotNetBridge and that the Encryption.dll assembly was loaded.

TestEncryption.prg (**Listing 10**) tests EncryptString and DecryptString, including a string previously encrypted with VFPEncryption.fll. Note that I do not recommend using SYS(2015) as the key for encryption and decryption, since it changes every time it's called; the same key has to be used to encrypt and decrypt a string. It's just used here for demo purposes.

Listing 10. TestEncryption.prg shows how EncryptString and DecryptString are called.

```
* Set up wwDotNetBridge.

do wwDotNetBridge
oBridge = GetwwDotNetBridge()
oBridge.LoadAssembly('Encryption.dll')

* Test encryption.

lcOriginal  = 'The quick brown fox'
lcKey       = sys(2015)
lcEncrypted = EncryptString(lcOriginal, lcKey)
lcDecrypted = DecryptString(lcEncrypted, lcKey)
messagebox('Original: ' + lcOriginal + chr(13) + ;
   'Encrypted: ' + lcEncrypted + chr(13) + ;
   'Decrypted: ' + lcDecrypted)

* Test decrypting a string previously encrypted with VFPEncryption.dll.

lcKey       = '_5100P6KGM'
lcEncrypted = '†ËM*¯Äp$ŸêvQ9:i€Á»Ÿ'
lcDecrypted = DecryptString(lcEncrypted, lcKey)
messagebox('Original: ' + lcOriginal + chr(13) + ;
   'Encrypted: ' + lcEncrypted + chr(13) + ;
   'Decrypted: ' + lcDecrypted)
```

Encryption.cs could be enhanced to add additional encryption-related functionality, such as hashing and checksum calculations if desired.

Email

Being able to send an email from an application is often very useful: emailing error information to the developer, sending invoices to customers, emailing payroll stubs to employees, etc. As with encryption, I've used a third-party library (West Wind Client Tools) for many years to provide email services to my applications. However, the version of the library I had didn't support using SSL to send emails (newer versions do but I needed this feature before that was supported), something that's required if you want to use Gmail or other servers that require it. So, once again, I decided to create a small .NET library that provides the features I need for sending email from a VFP application.

Listing 11 shows the code for SMTPLibrary.cs. The SMTP class in this file has two public methods: AddAttachment, which adds the specified filename as an attachment, and SendMail, which sends the email using the properties shown in **Table 1**.

Listing 11. SMTPLibrary.cs provides email capabilities.

```
using System;
using System.Collections.Generic;
using System.Net;
using System.Net.Mail;

namespace SMTPLibrary
{
    /// <summary>
    /// Sends email via SMTP.
    /// </summary>
    public class SMTP
    {
        public string MailServer { get; set; } = "";
        public int ServerPort { get; set; } = 25;
        public string SenderEmail { get; set; } = "";
        public string SenderName { get; set; } = "";
        public string Recipients { get; set; } = "";
        public string CCRecipients { get; set; } = "";
        public string BCCRecipients { get; set; } = "";
        public string Subject { get; set; } = "";
        public string Message { get; set; } = "";
        public string UserName { get; set; } = "";
        public string Password { get; set; } = "";
        public bool UseHtml { get; set; }
        public bool UseSsl { get; set; }
        public int Timeout { get; set; } = 30;
        public string ErrorMessage { get; private set; } = "";
        private List<string> _attachments = new List<string>();

        public bool SendMail()
        {
            // Set up the host.
            NetworkCredential basicCredential = new NetworkCredential(UserName,
                Password);
            SmtpClient smtpClient = new SmtpClient();
            smtpClient.Host = MailServer;
            smtpClient.UseDefaultCredentials = false;
            smtpClient.Credentials = basicCredential;
            smtpClient.Timeout = Timeout * 1000;
            smtpClient.Port = ServerPort;
            smtpClient.EnableSsl = UseSsl;
```

```
            // Set up the mail message.
            MailMessage message = new MailMessage();
            message.From = new MailAddress(SenderEmail, SenderName);
            message.Sender = message.From;
            message.IsBodyHtml = UseHtml;
            message.Body = Message;
            message.Subject = Subject;

            // Handle addresses.
            foreach (var address in Recipients.Split(new[] { ";" },
                StringSplitOptions.RemoveEmptyEntries))
            {
                message.To.Add(address);
            }
            foreach (var address in CCRecipients.Split(new[] { ";" },
                StringSplitOptions.RemoveEmptyEntries))
            {
                message.CC.Add(address);
            }
            foreach (var address in BCCRecipients.Split(new[] { ";" },
                StringSplitOptions.RemoveEmptyEntries))
            {
                message.Bcc.Add(address);
            }

            // Handle attachments.
            foreach (string attachment in _attachments)
            {
                message.Attachments.Add(new Attachment(attachment));
            }

            // Try to send the message.
            bool result = false;
            ErrorMessage = "";
            try
            {
                smtpClient.Send(message);
                result = true;
            }
            catch (Exception ex)
            {
                ErrorMessage = ex.Message;
            }
            message.Dispose();
            return result;
        }

        public void AddAttachment(string fileName)
        {
            _attachments.Add(fileName);
        }
    }
}
```

Table 1. The properties of the SMTP class.

Property	Type	Description
MailServer	String	The mail server address.
UserName	String	The user name for the server.
Password	String	The password for the server.
ServerPort	Int	The port to use (the default is 25).
SenderEmail	String	The email address for the sender.
SenderName	String	The name of the sender.
Recipients	String	A semi-colon delimited list of recipients.
CCRecipients	String	A semi-colon delimited list of CC recipients.
BCCRecipients	String	A semi-colon delimited list of BCC recipients.
Subject	String	The subject.
Message	String	The body of the message.
UseHtml	Bool	True if the body contains HTML.
UseSsl	Bool	True to use SSL.
Timeout	Int	The timeout in seconds (the default is 30).
ErrorMessage	String	The text of any error that occurred.

TestEmail.prg (**Listing 12**) shows how to use the SMTP class, in this case specifically with Gmail. For demo purposes, this code gets the password from an encrypted text file; in a real application, it would likely be stored in a table or some other location (although also encrypted for security; hence loading Encryption.dll). The email sent by this program contains formatted HTML and has an image file as an attachment.

Listing 12. TestEmail.prg shows how the SMTP class is used.

```
* Set up wwDotNetBridge.

do wwDotNetBridge
oBridge = GetwwDotNetBridge()
oBridge.LoadAssembly('Encryption.dll')
oBridge.LoadAssembly('SMTPLibrary.dll')

* Send an email.

loMail               = oBridge.CreateInstance('SMTPLibrary.SMTP')
loMail.MailServer    = 'smtp.gmail.com'
loMail.Username      = 'doug.o.hennig@gmail.com'
loMail.Password      = DecryptString(filetostr('DontDeploy\password.txt'), ;
   filetostr('DontDeploy\key.txt'))
loMail.ServerPort    = 587
loMail.SenderEmail   = 'doug.o.hennig@gmail.com'
```

```
loMail.SenderName    = 'Doug Hennig'
loMail.Recipients    = 'dhennig@stonefield.com'
loMail.Subject       = 'Test email'
loMail.Message       = 'This is a test message. ' + ;
   '<strong>This is bold text</strong>. ' + ;
   '<font color="red">This is red text</font>'
loMail.UseSsl        = .T.
loMail.UseHtml       = .T.
loMail.AddAttachment('koko.jpg')
llReturn = loMail.SendMail()
if not llReturn
   messagebox(loMail.ErrorMessage)
endif not llReturn
```

Sending SMS messages

Some applications, such as scheduling programs for medical offices or hair stylists, need to send text (SMS) messages to mobile devices. Although there are services such as Twilio that can do this (and there's a VFPX project, https://github.com/VFPX/TwilioX, that makes it easy to use Twilio from VFP), a simple way to send an SMS message is to send an email to the user's carrier prefixed with the mobile device number and "@" (for example 2049995555@mycarrier.com). That requires knowing the email address of the carrier. Fortunately, a GitHub project (https://github.com/cubiclesoft/email_sms_mms_gateways) provides a list of email addresses for carriers as a JSON list. So, sending an SMS message means determining the email address to use for a specific carrier by downloading the latest version of the JSON file, converting the JSON to something VFP can easily access, finding the carrier, and then sending an email to their address. All three of these capabilities (downloading, converting JSON, and sending emails) are built into .NET, so I created a wrapper program to make it easy to call from VFP. I adapted the wrapper from code written by Rod Stephens (https://tinyurl.com/ybmr3hmy).

Listing 13 shows the code for SMS.cs. The main method in the SMSLibrary class in this file is SendMessage. To call it, set the properties shown in **Table 2** and pass it the name and mobile number of the recipient, the carrier's email address, and the subject and body of the message.

Listing 13. SMS.cs provides SMS capabilities.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net;
using System.Net.Mail;
using System.Web.Script.Serialization;

namespace SMS
{
    /// <summary>
    /// Adapted from http://csharphelper.com/blog/2018/01/how-to-send-an-sms-text-
    /// message-in-c/
    /// </summary>
    public class SMSLibrary
    {
        public string ErrorMessage { get; set; } = "";
        public string SenderName { get; set; }
```

```
public string SenderEmail { get; set; }
public string Password { get; set; }
public string Server { get; set; }
public int Port { get; set; } = 25;
public bool UseSSL { get; set; } = false;
public string[] Countries
{
    get
    {
        if (_countryInfos.Count == 0)
        {
            GetInfo();
        }
        return _countryInfos.Select(c => c.Value.CountryName).ToArray();
    }
}

private Dictionary<string, CountryInfo> _countryInfos =
     new Dictionary<string, CountryInfo>();

/// <summary>
/// Returns a list of carriers for the specified country.
/// </summary>
/// <param name="country">
/// The country to get carriers for.
/// </param>
/// <returns>
/// An array of carriers for the specified country or a blank array for
/// a non-existent country.
/// </returns>
public string[] GetCarriersForCountry(string country)
{
    string[] carriers;
    try
    {
        CountryInfo countryInfo = _countryInfos[country];
        carriers = countryInfo.Carriers.Select(c => c.Key).ToArray();
    }
    catch (Exception)
    {
        ErrorMessage = "Country not found";
        carriers = new string[0];
    }
    return carriers;
}

/// <summary>
/// Returns a list of emails for the specified carrier in the specified
/// country.
/// </summary>
/// <param name="country">
/// The country the carriers belongs to.
/// </param>
/// <param name="carrier">
/// The carrier to get emails for.
/// </param>
/// <returns>
/// An array of emails for the specified carrier in the specified country
/// or a blank array for a non-existent country or carrier.
/// </returns>
public string[] GetEmailsForCarrier(string country, string carrier)
{
```

```
    string[] emails;
    try
    {
        CountryInfo countryInfo = _countryInfos[country];
        try
        {
            CarrierInfo carrierInfo = countryInfo.Carriers[carrier];
            emails = carrierInfo.Emails.ToArray();
        }
        catch (Exception)
        {
            ErrorMessage = "Carrier not found";
            emails = new string[0];
        }
    }
    catch (Exception)
    {
        ErrorMessage = "Country not found";
        emails = new string[0];
    }
    return emails;
}

/// <summary>
/// Send an SMS message.
/// </summary>
/// <param name="name">
/// The recipient's name.
/// </param>
/// <param name="phone">
/// The recipient's phone number.
/// </param>
/// <param name="subject"></param>
/// <param name="body"></param>
public void SendMessage(string name, string phone, string carrierEmail,
     string subject, string body)
{
    // Strip unwanted characters from the phone number and create the
    // email address to use.
    string toPhone = phone.Trim().Replace("-", "").
        Replace("(", "").Replace(")", "").Replace("+", "");
    string toEmail = toPhone + "@" + carrierEmail;

    // Create the mail message.
    MailAddress from_address = new MailAddress(SenderEmail, SenderName);
    MailAddress to_address = new MailAddress(toEmail, name);
    MailMessage message = new MailMessage(from_address, to_address);
    message.Subject = subject;
    message.Body = body;

    // Get the SMTP client.
    SmtpClient client = new SmtpClient()
    {
        Host = Server,
        Port = Port,
        EnableSsl = UseSSL,
        UseDefaultCredentials = false,
        Credentials = new NetworkCredential(from_address.Address,
            Password),
    };

    // Send the message.
```

```
            client.Send(message);
        }

        /// <summary>
        /// Gets the SMS information for the library.
        /// </summary>
        private void GetInfo()
        {
            // Get the data file.
            const string url =
"https://raw.github.com/cubiclesoft/email_sms_mms_gateways/master/sms_mms_gateways
.txt";
            string serialization = GetTextFile(url);

            // Deserialize it.
            JavaScriptSerializer serializer = new JavaScriptSerializer();
            Dictionary<string, object> dict = (Dictionary<string,
                object>)serializer.DeserializeObject(serialization);

            // Get the countries.
            Dictionary<string, object> countries =
                (Dictionary<string, object>)dict["countries"];
            Dictionary<string, CountryInfo> countriesByAbbrev =
                new Dictionary<string, CountryInfo>();
            foreach (KeyValuePair<string, object> pair in countries)
            {
                CountryInfo country_info = new CountryInfo()
                    { CountryAbbreviation = pair.Key,
                       CountryName = (string)pair.Value };
                _countryInfos.Add(country_info.CountryName, country_info);
                countriesByAbbrev.Add(country_info.CountryAbbreviation,
                    country_info);
            }

            // Get the SMS carriers.
            Dictionary<string, object> sms_carriers =
                (Dictionary<string, object>)dict["sms_carriers"];
            foreach (KeyValuePair<string, object> pair in sms_carriers)
            {
                // Get the corresponding CountryInfo.
                CountryInfo country_info = countriesByAbbrev[pair.Key];

                // Get the country's carriers.
                Dictionary<string, object> carriers =
                    (Dictionary<string, object>)pair.Value;
                foreach (KeyValuePair<string, object> carrier_pair in carriers)
                {
                    // Create a CarrierInfo for this carrier.
                    CarrierInfo carrier_info = new CarrierInfo()
                       { CarrierAbbreviation = carrier_pair.Key };
                    object[] carrier_values = (object[])carrier_pair.Value;
                    carrier_info.CarrierName = (string)carrier_values[0];
                    for (int email_index = 1; email_index < carrier_values.Length;
                       email_index++)
                    {
                        string email = (string)carrier_values[email_index];
                        carrier_info.Emails.Add(email.Replace("{number}@", ""));
                    }
                    country_info.Carriers.Add(carrier_info.CarrierName,
                       carrier_info);
                }
            }
```

```
        }

        /// <summary>
        /// Get the text file at the specified URL.
        /// </summary>
        /// <param name="url">
        /// The URL for the file to retrieve.
        /// </param>
        /// <returns>
        /// The content of the file or an empty string if it failed.
        /// </returns>
        private string GetTextFile(string url)
        {
            string result = "";
            try
            {
                url = url.Trim();
                if (!url.ToLower().StartsWith("http"))
                {
                    url = "http://" + url;
                }
                WebClient web_client = new WebClient();
                MemoryStream image_stream =
                    new MemoryStream(web_client.DownloadData(url));
                StreamReader reader = new StreamReader(image_stream);
                result = reader.ReadToEnd();
                reader.Close();
            }
            catch (Exception ex)
            {
                ErrorMessage = "Error downloading file " + url + '\n' +
                    ex.Message;
            }
            return result;
        }
    }
}
```

Table 2. The properties of the SMSLibrary class.

Property	Type	Description
Server	String	The mail server address.
Password	String	The password for the server.
Port	Int	The port to use (the default is 25).
SenderEmail	String	The email address for the sender.
SenderName	String	The name of the sender.
UseSSL	Bool	True to use SSL.
Countries	String[]	An array of valid countries.
ErrorMessage	String	The text of any error that occurred.

There are a couple of helper methods:

- GetCarriersForCountry: access the Countries property to get a list of supported countries, then call GetCarriersForCountry, passing it the name of the country to get a list of carriers for.
- GetEmailsForCarrier: pass this method the name of a country and a valid carrier in that country, obtained from the list returned by GetCarriersForCountry, to get a list of the email addresses for that carrier.

TestSMS.prg (**Listing 14**) shows how to use the SMS class, in this case specifically with Gmail. For demo purposes, this code gets the password from an encrypted text file; in a real application, it would likely be stored in a table or some other location (although also encrypted for security; hence loading Encryption.dll). This program shows the list of countries supported, the list of carriers for Canada, and the list of email addresses (there's actually only one) for my carrier. Note the code uses GetPropertyEx and InvokeMethod since the values are .NET arrays and these methods automatically convert them into something VFP can use (see https://tinyurl.com/y7xb2aub for details). It then sends a text message to my cell phone.

A production application would store the preferences for each person it sends a text message to. For example, there may be a form with a combobox of countries. When the user chooses a country, a combobox of carriers for that country is populated. When the user chooses a carrier, a combobox of email addresses for that carrier is populated. The user's choices are saved in that person's record.

Listing 14. TestSMS.prg shows how the SMS class is used.

```
* Set up wwDotNetBridge.

do wwDotNetBridge
oBridge = GetwwDotNetBridge()
oBridge.LoadAssembly('Encryption.dll')
oBridge.LoadAssembly('SMS.dll')

* Set the SMS properties.

loSMS             = oBridge.CreateInstance('SMS.SMSLibrary')
loSMS.Server      = 'smtp.gmail.com'
loSMS.Port        = 587
loSMS.SenderName  = 'Doug Hennig'
loSMS.SenderEmail = 'doug.o.hennig@gmail.com'
loSMS.Password    = DecryptString(filetostr('DontDeploy\password.txt'), ;
   filetostr('DontDeploy\key.txt'))
loSMS.UseSSL      = .T.

* Display a list of countries.

loCountries = oBridge.GetPropertyEx(loSMS, 'Countries')
lcCountries = ''
for lnI = 0 to loCountries.Count - 1
   lcCountry   = loCountries.Item(lnI)
   lcCountries = lcCountries + lcCountry + chr(13)
next lnI
messagebox(lcCountries, 0, 'Countries')

* Display the carriers for Canada.
```

```
loCarriers = oBridge.InvokeMethod(loSMS, 'GetCarriersForCountry', 'Canada')
lcCarriers = ''
for lnI = 0 to loCarriers.Count - 1
   lcCarrier  = loCarriers.Item(lnI)
   lcCarriers = lcCarriers + lcCarrier + chr(13)
next lnI
messagebox(lcCarriers, 0, 'Carriers for Canada')

* Get the emails for MTS.

loEmails = oBridge.InvokeMethod(loSMS, 'GetEmailsForCarrier', 'Canada', ;
   'Manitoba Telecom/MTS Mobility')
lcEmails = ''
for lnI = 0 to loEmails.Count - 1
   lcEmail  = loEmails.Item(lnI)
   lcEmails = lcEmails + lcEmail + chr(13)
next lnI
messagebox(lcEmails, 0, 'Emails for MTS')

* Send a text message.

lcPhone  = filetostr('DontDeploy\phone.txt')
llReturn = loSMS.SendMessage('Doug Hennig', lcPhone, ;
   'text.mtsmobility.com', 'FoxCon', 'SMS sent by VFP')
if not llReturn
   messagebox(loSMS.ErrorMessage)
endif not llReturn
```

Formatting strings

The VFP TRANSFORM() function converts Date, DateTime, numeric, and other values to an optionally formatted string. However, the formats available are somewhat limited, especially for Date and DateTime values; see **Table 3** for the formats available. To make matters more complicated, the value returned depends on the settings of SET CENTURY, SET HOURS, and SET MARK for Date/DateTime values (although not for all formats as you can see in Table 3) and SET POINT, SET SEPARATOR, SET CURRENCY, and SET DECIMALS for numeric values, except if SET SYSFORMATS ON is used, in which case all of those are ignored and the system settings are used (which is normally the safest approach).

Table 3. Date/DateTIme formats available.

Format	Description	Example (12 hours)	Example (24 hours)
@D	Converts to current SET DATE format	10/05/2017 02:25:00 PM	10/05/2017 14:25:00
@E	Converts to BRITISH date format	05/10/2017 02:25:00 PM	05/10/2017 14:25:00
@YL	Uses Long Date system setting	Thursday, October 5, 2017, 2:25:00 PM	same
@YS	Uses Short Date system setting	10/05/2017 2:25:00 PM	same

It's even more complicated if you just want to display the time: you have to use SUBSTR() to pull out just the time part, but you can't hard-code the starting position

because it depends on how the user's system formats dates. Instead, you have to use something like:

```
lcTime = substr(transform(ltValue), at(' ', transform(ltValue)) + 1)
```

.NET has a similar method to the TRANSFORM() function: ToString. However, ToString has a lot more formats available for Date, DateTime, and numeric values. For example, a format string of HH:mm:ss shows just the time part of a DateTime value in 24-hour format, such as 14:20:00. Also, it automatically respects the user's regional settings, so it displays day and month names in the correct language and uses the correct separators for thousands and decimal places. See the following URLs for the different formats available:

- Standard numeric format strings: http://tinyurl.com/y86clj9k
- Custom numeric format strings: http://tinyurl.com/ydgw5zml
- Standard datetime format strings: http://tinyurl.com/y7nfeook
- Custom datetime format strings: http://tinyurl.com/ycwh45af

Eric Selje wrote a StringFormat function (http://saltydogllc.com/string-format-for-visual-foxpro/) that simulates what ToString does. However, using wwDotNetBridge, we can call ToString directly.

FormatValue.prg (**Listing 15**) is a wrapper for ToString written by Rick Strahl. Pass it a value and optionally a format string to format the value as desired. Note that unlike DecryptString.prg and EncryptString.prg, FormatValue.prg doesn't require you to set up wwDotNetBridge first or load any assembly.

Listing 15. FormatValue.prg formats values as desired.

```
lparameters tuValue, ;
   tcFormatString
local loBridge, ;
   lcResult
if isnull(tuValue)
   return 'null'
endif isnull(tuValue)
do wwDotNetBridge
loBridge = GetwwDotNetBridge()
if empty(tcFormatString)
   lcResult = loBridge.InvokeMethod(tuValue, 'ToString')
else
   lcResult = loBridge.InvokeMethod(tuValue, 'ToString', tcFormatString)
endif empty(tcFormatString)
return lcResult
```

TestFormatValue.prg (**Listing 16**) shows how FormatValue works with both DateTime and numeric values.

Listing 16. TestFormatValue.prg shows how FormatValue works.

```
* First show VFP date/datetime formats.

ltNow = datetime(2017, 10, 5, 14, 20, 0)
lcMessage = 'No format: ' + transform(ltNow) + chr(13) + ;
   '@D: ' + transform(ltNow, '@D') + chr(13) + ;
```

```
    '@E: ' + transform(ltNow, '@E') + chr(13) + ;
    '@YL: ' + transform(ltNow, '@YL') + chr(13) + ;
    '@YS: ' + transform(ltNow, '@YS')
messagebox(lcMessage, 0, 'VFP formats')

* Now show .NET date/datetime formats.

lcMessage = 'No format: ' + FormatValue(ltNow) + chr(13) + ;
    'MMM dd, yyyy: ' + FormatValue(ltNow, 'MMM dd, yyyy') + chr(13) + ;
    'MMMM dd, yyyy: ' + FormatValue(ltNow, 'MMMM dd, yyyy') + chr(13) + ;
    'HH:mm:ss: ' + FormatValue(ltNow, 'HH:mm:ss') + chr(13) + ;
    'h:m:s tt: ' + FormatValue(ltNow, 'h:m:s tt') + chr(13) + ;
    'MMM d @ HH:mm: ' + FormatValue(ltNow, 'MMM d @ HH:mm') + chr(13) + ;
    'r (RFC format): ' + FormatValue(ltNow, 'r') + chr(13) + ;
    'u (Universal sortable): ' + FormatValue(ltNow, 'u')
messagebox(lcMessage, 0, '.NET formats')

* .NET numeric formats.

lnValue = 1233.2255
lnInt   = int(lnValue)
lcMessage = 'No format: ' + FormatValue(lnValue) + chr(13) + ;
    'c2: ' + FormatValue(lnValue, 'c2') + chr(13) + ;
    'n2: ' + FormatValue(lnValue, 'n2') + chr(13) + ;
    'd7: ' + FormatValue(lnInt, 'd7') + chr(13) + ;
    'p1: ' + FormatValue(lnInt/1000, 'p1')
messagebox(lcMessage, 0, '.NET formats')
```

Another utility by Rick, FormatString.prg (**Listing 17**), makes it easy to replace placeholders in a string with specified values. For example, to display information about an error in an error handler, you might use code like this (taken from TestFormatString.prg):

```
lcMessage = 'Error #' + transform(tnError) + ' occurred in line ' + ;
   transform(tnLineNo) + ' of ' + justfname(tcProgram) + ' on ' + ;
   transform(datetime())
```

That requires concatenating numerous strings and having to use TRANSFORM() on certain values if necessary. Compare that with the following, which uses FormatString:

```
lcMessage = FormatString('Error #{0} occurred in line {1} of {2} on ' + ;
   '{3:MMMM d, yyyy HH:mm:ss}', tnError, tnLineNo, justfname(tcProgram), ;
   datetime())
```

No need to worry about whether a value is character or not; FormatString handles the conversion automatically.

FormatString.prg uses the static Format method of the .NET String class to do the work. Format expects a format string containing placeholders such as {0} for the first one (since .NET is zero-based), {1} for the second, and so on. Placeholders can contain a format string, such as {3:MMMM d, yyyy HH:mm:ss} to format a DateTime value as specified. Format has numerous overloads that accept a different number of parameters, so the code uses a CASE statement to pass only those parameters actually passed to the function to the Format method.

Listing 17. FormatString.prg replaces placeholders in a string (some code omitted for brevity).

```
lparameters tcFormat, ;
   tuValue1, ;
```

```
   tuValue2, ;
   tuValue3, ;
   tuValue4, ;
   tuValue5, ;
   tuValue6, ;
   tuValue7, ;
   tuValue8, ;
   tuValue9, ;
   tuValue10
local lnParms, ;
   loBridge
lnParms = pcount()
do wwDotNetBridge
loBridge = GetwwDotNetBridge()
do case
   case lnParms = 2
      return loBridge.InvokeStaticMethod('System.String', 'Format', ;
         tcFormat, tuValue1)
   case lnParms = 3
      return loBridge.InvokeStaticMethod('System.String', 'Format', ;
         tcFormat, tuValue1, tuValue2)
   case lnParms = 4
      return loBridge.InvokeStaticMethod('System.String', 'Format', ;
         tcFormat, tuValue1, tuValue2, tuValue3)
   case lnParms = 5
      return loBridge.InvokeStaticMethod('System.String', 'Format', ;
         tcFormat, tuValue1, tuValue2, tuValue3, tuValue4)
* Rest of cases omitted for brevity
   otherwise
      throw 'Too many parameters for FormatString'
endcase
```

"Humanizing" strings

There may be times when you need to format text in more friendly, "humanized" ways. For example, you may wish to display the steps in a process as ordinals: first, second, third, or 1st, 2nd, 3rd, etc. While you can write some VFP code to handle this for you in one language, it becomes much more cumbersome if you need to handle multiple languages.

The Humanizer .NET library handles many types of humanizing tasks such as:

- Converting Date and DateTime values to expressions such as "yesterday" or "2 hours from now."
- Converting numbers to words, such as "fifteen" for 15.
- Converting nouns to their plural or singular equivalents. It's smart enough to know that the plural of an English word doesn't always have an "s" suffix; for example, it knows "people" is the plural of "person."

Humanizer is also culture-sensitive, so it knows not only the correct language to use but also the rules of that language.

Humanizer methods are implemented as extension methods. That means they extend a data type and make it appears as the methods belong to that type. For example, in **Figure 6**, you can see that IntelliSense in Visual Studio indicates that there's a ToOrdinalWords method available for integers. That isn't a built-in .NET method; it's provided by the Humanizer library.

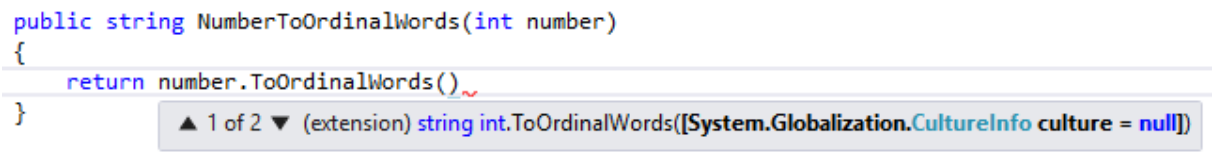

Figure 6. Humanizer methods are implemented as extension methods.

Extension methods can't be called from wwDotNetBridge, so Rick created a wrapper class named FoxHumanizer that provides some of the Humanizer methods. I've extended the class a bit, adding some additional methods and removing a couple that I didn't find that useful (such as TruncateString) because VFP provides that functionality natively.

FoxHumanizer has regular methods rather than static ones, which means the class has to be instantiated to use it. That's simply so the syntax for calling the methods is a little cleaner:

```
loHuman = loBridge.CreateInstance('FoxHumanizer.FoxHumanizer')
loHuman.Method(value)
```

instead of:

```
loBridge.InvokeStaticMethod('FoxHumanizer.FoxHumanizer', 'Method', value)
```

The FoxHumanizer class is shown in **Listing 18**. This class only wraps some of the functionality of the Humanizer library; see https://github.com/Humanizr/Humanizer for details on the capabilities of this library.

Listing 18. The FoxHumanizer class is a wrapper for the Humanizer library.

```
using System;
using Humanizer;
namespace FoxHumanizer
{
    public class FoxHumanizer
    {
        public string HumanizeDate(DateTime date)
        {
            return date.Humanize(false);
        }

        public string HumanizeDateForCulture(DateTime date,
          System.Globalization.CultureInfo culture)
        {
            return date.Humanize(false, null, culture);
        }

        public string Humanize(string text)
        {
            return text.Humanize();
        }

        public string NumberToWords(int number)
        {
            return number.ToWords();
        }

        public string NumberToOrdinal(int number)
        {
```

```
            return number.Ordinalize();
        }

        public string NumberToOrdinalWords(int number)
        {
            return number.ToOrdinalWords();
        }

        public string ToQuantity(string single, int qty)
        {
            return single.ToQuantity(qty, ShowQuantityAs.Words);
        }

        public string Pluralize(string single)
        {
            return single.Pluralize(true);
        }

        public string Singularize(string pluralized)
        {
            return pluralized.Singularize(true);
        }

        public string ToByteSize(int byteSize)
        {
            return byteSize.Bytes().Humanize("#.##");
        }
    }
}
```

Obviously, this class requires the Humanizer library. The simplest way to get a copy of that library is to choose NuGet Package Manager, Manage NuGet Packages for Solution… from the Tools menu in Visual Studio, search for Humanizer, and install it. When you deploy an application using FoxHumanizer, you have to include FoxHumanizer.dll, Humanizer.dll, plus folders containing resources for the cultures your users might use (for example, the "de" folder contains the resource for German).

TestHumanizer.prg (**Listing 19**) shows how the various methods in FoxHumanizer work.

Listing 19. TestHumanizer.prg shows how FoxHumanizer works.

```
* Set up wwDotNetBridge and instantiate the FoxHumanizer class.

do wwDotNetBridge
loBridge = GetwwDotNetBridge()
loBridge.LoadAssembly('FoxHumanizer.dll')
loHuman = loBridge.CreateInstance('FoxHumanizer.FoxHumanizer')

* Humanize dates.

messagebox('date() - 1: ' + loHuman.HumanizeDate(date() - 1) + chr(13) + ;
   'datetime() - 2 * 60 * 60: ' + ;
      loHuman.HumanizeDate(datetime() - 2 * 60 * 60) + chr(13) + ;
   'gomonth(date(), -1): ' + loHuman.HumanizeDate(gomonth(date(), -1)))

* Handle a non-current culture (German in this case).

loCulture = loBridge.CreateInstance('System.Globalization.CultureInfo', ;
   'de-DE')
```

```
messagebox('For German, date() - 1 displays as ' + ;
   loHuman.HumanizeDateForCulture(date() - 1, loCulture))

* Humanize numbers.

messagebox('NumberToOrdinal(3): ' + loHuman.NumberToOrdinal(3) + chr(13) + ;
   'NumberToOrdinalWords(3): ' + loHuman.NumberToOrdinalWords(3) + chr(13) + ;
   'ToByteSize(13122): ' + loHuman.ToByteSize(13122) + chr(13) + ;
   'ToByteSize(1221221): ' + loHuman.ToByteSize(1221221) + chr(13) + ;
   'NumberToWords(15): ' + loHuman.NumberToWords(15))

* NumberToWords only handles integers so we'll split up dollars and cents.

lnAmount  = 12345.67
lnDollars = int(lnAmount)
lnCents   = (lnAmount - lnDollars) * 100
messagebox('12345.67: ' + FormatString('{0} dollars and {1} cents', ;
   loHuman.NumberToWords(lnDollars), loHuman.NumberToWords(lnCents)))

* Humanize strings.

messagebox("ToQuantity('dollar', 3): " + loHuman.ToQuantity('dollar', 3) + ;
      chr(13) + ;
   "Pluralize('speaker'): " + loHuman.Pluralize('speaker') + chr(13) + ;
   "Pluralize('person'): " + loHuman.Pluralize('person') + chr(13) + ;
   "Singularize('mice'): " + loHuman.Singularize('mice') + chr(13) + ;
   "Humanize('PascalCaseString'): " + loHuman.Humanize('PascalCaseString') + ;
      chr(13) + ;
   "Humanize('Underscored_String'): " + ;
      loHuman.Humanize('Underscored_String'))
```

References

Here are links to articles and documentation about wwDotNetBridge:

- wwDotNetBridge home page: http://www.west-wind.com/wwDotnetBridge.aspx
- wwDotNetBridge documentation: http://tinyurl.com/ltagjhk
- wwDotNetBridge white paper: http://tinyurl.com/lclaflx
- "Calling async/await .NET methods with wwDotnetBridge": http://tinyurl.com/yd2wlpa8

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).