Calling FoxPro COM objects from ASP.NET Revisited

By Rick Strahl
www.west-wind.com

Microsoft's .NET framework has now been out for quite a few years and ASP.NET has become Microsoft's flagship Web technology. Now in Version 4.5 ASP.NET has proven itself as a powerful, extremely flexible and highly scalable platform for building Web applications.

I've written about how to do COM Interop with ASP.NET for many years, and this article is a re-evaluating ASP.NET more than 10 years after its initial creation. I'll also cover some of the basics, but the focus is on using newer features and integrating with ASP.NET MVC. My existing older article covers the basics and overview of how COM Interop works and how you can apply it:

http://west-wind.com/presentations/VfpDotNetInterop/aspcominterop.aspx

Most of the material in that article is still as applicable today as it was when .NET 1.0 was released and the original article was written.

In this article I'll cover what's new and improved and show how COM Interop development with .NET has gotten a lot easier with .NET 4.0 and the use of .NET dynamics. In this article, I'll cover using HTML development with ASP.NET MVC and focus on using FoxPro COM objects as business objects that drive the HTML content created. This is a very long document that covers a step by step walkthrough with a lot of screen shots and complete code listings. But first here's a short review of the basics of how COM Interop works.

ASP.NET and FoxPro

Visual FoxPro doesn't quite fit into the .NET world, given that FoxPro is not one of the .NET supported languages like C# or Visual Basic. Although .NET does not allow direct interaction with Visual FoxPro code, .NET can call COM components including those built with Visual FoxPro fairly easily. From the very beginning of .NET COM Interop was supported.

The original version used the Runtime Callable Wrapper (RCW) which allowed importing of COM type libraries and map them to .NET types that provided a simulation of a .NET class for a COM object. This process worked Ok, but there were some problems with keeping the typelibrary and COM Interop type in sync and the fact that FoxPro has very limited type library export functionality to express object hierarchies properly. FoxPro only supports flat objects in type libraries – no support for nested object properties.

Luckily in .NET 4.0 the *dynamic* type was introduced, which greatly simplifies COM Interop by allowing COM objects to be treated like untyped objects. Dynamic behavior basically gives you the same functionality you have in Visual FoxPro where you can create an instance of a COM object and immediately access any of its properties, without having to map the object to some sort of .NET type first.

Calling VFP COM Components from .NET

COM Interop is achieved in .NET through an Interop layer that sits between .NET and the managed COM component. .NET runs what is known as 'managed code', which is code that is hosted in the .NET Runtime. The .NET Common Language Runtime (CLR) provides a type system and the operational rules of the system. This system does not run on native machine code and in order to call out to 'unmanaged' or native code like Windows API calls or COM calls have to go through an interoperability layer.

For COM Interop there are two of Interop layers available, one for hosting COM components in .NET called a Runtime Callable Wrapper (RCW) and one for hosting .NET components in non .NET COM clients called the COM Callable Wrapper (CCW). In this article I'll cover only the Runtime Callable Wrapper for COM Interop from ASP.NET to FoxPro COM components.

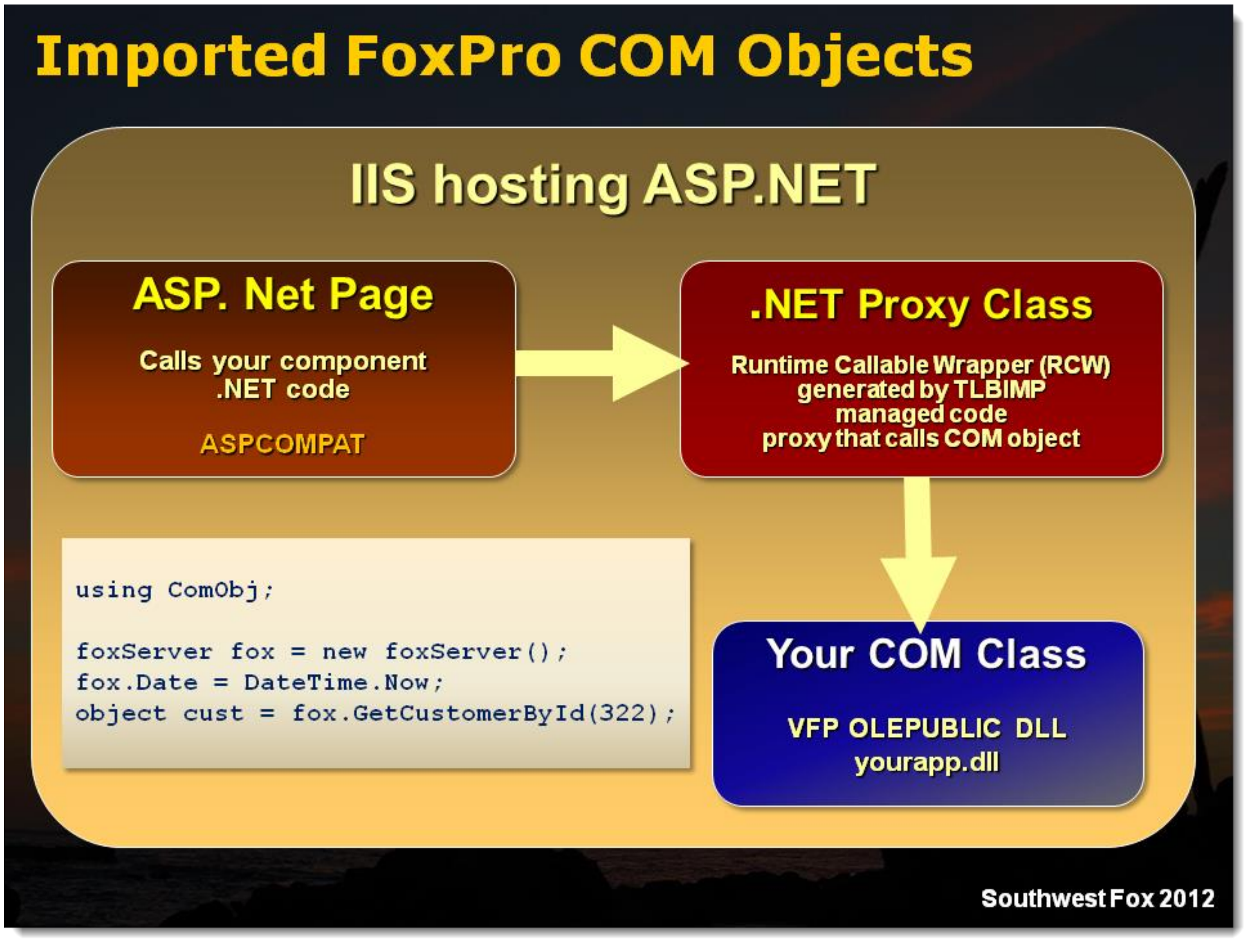
There are two ways to create use COM objects in .NET:

- Early Binding via Type Library imports
- Late Binding via the *dynamic* type or by using .NET Reflection

COM Type Library Imports – not recommended anymore

Type library imports are accomplished by using the TLBIMP.exe component from the .NET SDK. If you're using Visual Studio you can simply navigate to a COM object DLL or .TLB file import it. .NET then creates an Interop assembly with one or more .NET interfaces and classes that map the type library's interfaces and classes which is linked to the current project automatically.

The imported type uses early binding where .NET instantiates the COM object and manages the lifetime and member access through these generated COM Interop types transparently through fixed virtual COM interface mapping. I discussed type library imports extensively in my older article and you can check there for more information.



```
using ComObj;

foxServer fox = new foxServer();
fox.Date = DateTime.Now;
object cust = fox.GetCustomerById(322);
```

Figure 1: *When calling a COM component through an imported Interop assembly, .NET marshals the call through the Interop Assembly.The Runtime Callable Wrapper exposes a simple .NET method/property and handles the call against the FoxPro COM component.*

Typelibrary Imports are problematic

While the process looks really straight forward and is efficient with stable COM components that are fully functional and don't change frequently, the process can be problematic if the COM object is developed alongside the Web application and changes frequently. In Visual FoxPro especially it's problematic because VFP can be a bit schizophrenic when it comes to consistently maintain member name casing when compiling COM objects, which can screw up .NET type names when re-importing after making a change in the COM objects interface signature.

FoxPro also has a more general problem with type libraries: FoxPro COM objects can't express complex object hierarchies. FoxPro exported type libraries can't reference nested types on child properties. Any child objects – even if declared with proper COMATTRIB type attributes – only generate generic Variant export types. This means if you export a class that has child objects you can access the base members, but any child members simply come up as untyped object types in .NET which defeats some of the benefits of type library imports in the first place.

There are also problems with proper type mapping – because typelibrary imports happen at compile time when importing a type library the runtime can't change the type of a property at runtime. For some types like FoxPro Collections or Objects this can result in invalid type mappings that can make properties inaccessible.

There are other issues caused by type library type mismatches between FoxPro's type exports and what the value actually contains. For working with FoxPro type library imports in .NET tend to be too finicky, unless you plan on using a very stable component that won't be changing much.

We won't be discussing type library binding in this article as I covered it in the old article.

Late Binding with .NET Dynamics and Reflection

In this article I'll use Late Binding with dynamic types for all examples, because it's simply easier to work with and produces much more reliable behavior. Dynamic types are new in .NET 4.0.

When you use a COM object in Visual FoxPro, you generally use late binding: You use CREATEOBJECT() to create a COM instance and then fire away at the members of the object using the IDispatch interface, which is a late bound interface that discovers and accesses members at runtime as you are calling them. This runtime binding allows more flexibility in the API as types are not statically bound – it provides more flexibility in code.

Late binding was possible in .NET prior to .NET 4.0, but it required some pretty ugly code using .NET Reflection. Helper methods made this easier but the syntax was still kind of ugly and made code more difficult to read and write. I covered Reflection in the old article since it was written pre-.NET 4.0. Most of the Reflection code is no longer necessary in .NET 4.0 because dynamic types basically handle the nasty Reflection code behind the scenes.

With .NET 4.0 Microsoft introduced the Dynamic Language Runtime, which provides more natural late binding syntax using familiar object.member syntax against any 'untyped' .NET class. To .NET, COM objects are untyped objects since they don't have .NET type information. The DLR enables runtime type discovery for .NET types as well as COM objects and so you effectively get the same behavior you get when using COM object in FoxPro using runtime late binding.

The result is that you can use familiar . syntax to walk through a COM object even if there's no .NET type imported in the first place. So you can receive a Customer instance and reference Customer.Address.PhoneNumber:

C# - Creating and calling a COM object with .NET *dynamic* types

```
// create a COM object -
// using a ComHelper as there's no CreateObject() in C#
dynamic fox = ComHelper.CreateObject("firstFox.WebLogServer");
```

```
// create a FoxPro business object
dynamic entry = fox.ComHelper.CreateObject("blog_entry");

// Call Load() method on FoxPro business object
if (!entry.Load(id))
   throw new ApplicationException("Invalid Id passed.");

string title = entry.oData.Title;
DateTime entered = entry.oData.Entered;
```

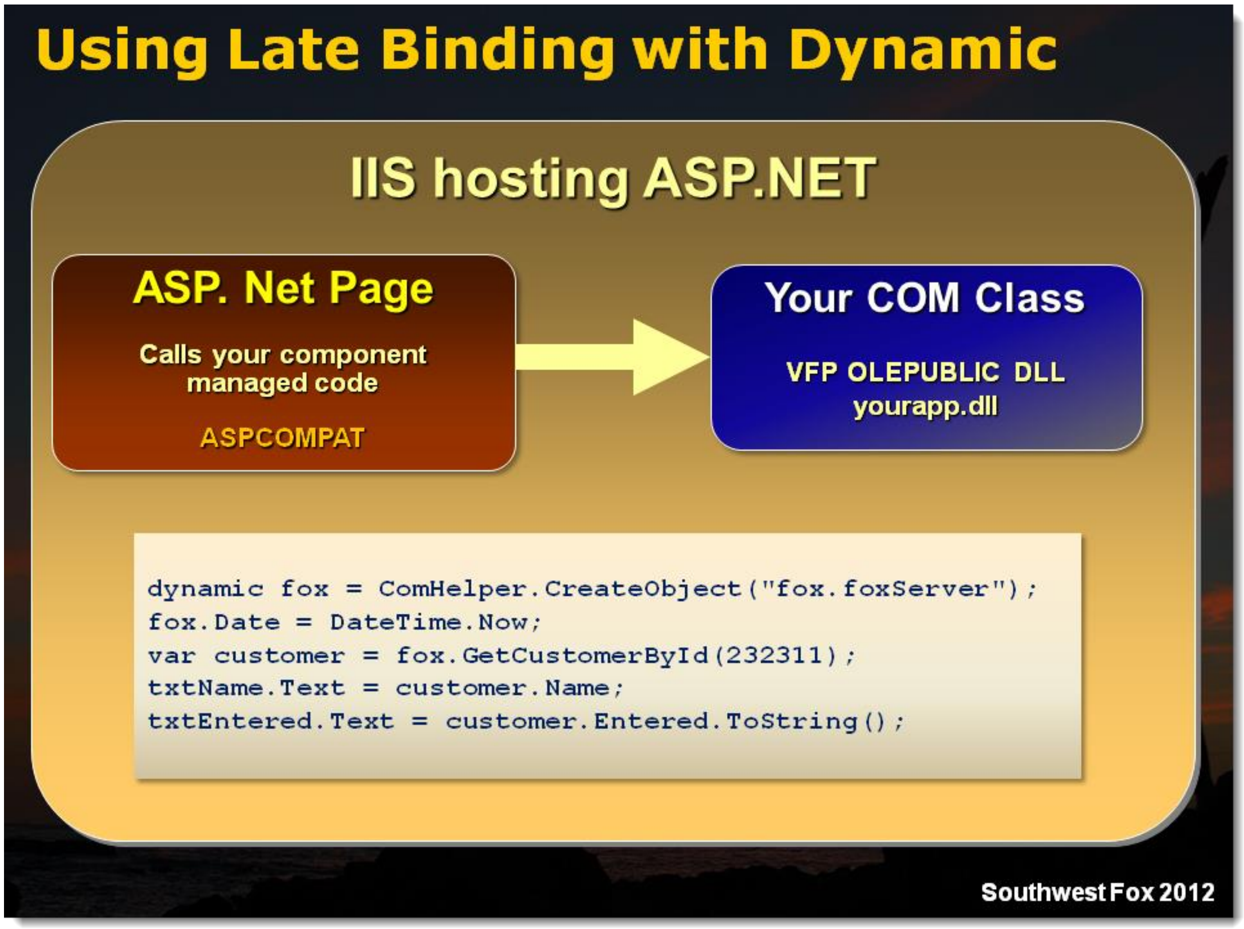
The syntax should be very familiar to a Fox developer – it's the same thing you would use to access that same COM object in FoxPro. The dynamic type is a loosely typed value in .NET that 'dynamically' retrieves the value from the COM object at runtime. Since the value is dynamic there's no strong typing involved – and no Intellisense on the COM object properties.

Note that in order to instantiate a COM object you need to use Reflection since C# doesn't not have a mechanism to create a COM object in the language (VB.NET has CreateObject() however). The ComHelper.CreateObject() method (provided in the samples) provides this functionality in a static function like this:

C# - Instantiating a COM object by ProgId via Reflection

```
public class ComHelper
{
    public static dynamic CreateObject(string progId)
    {
        Type type = Type.GetTypeFromProgID(progId);
        if (type == null)
            return null;

        return Activator.CreateInstance(type);
    }
    …
}
```

From a developer perspective using dynamics and late binding is a more direct path to COM object access as there's no intermediate proxy class that needs to be generated or updated when the type library changes. You simply create a COM instance or receive a COM object from a method call or property and then access its property members just as you do in FoxPro.



```
dynamic fox = ComHelper.CreateObject("fox.foxServer");
fox.Date = DateTime.Now;
var customer = fox.GetCustomerById(232311);
txtName.Text = customer.Name;
txtEntered.Text = customer.Entered.ToString();
```

Figure 2 *– Late binding with dynamic lets you use the familiar object.member.childmember syntax without an intermediate object. It works similar to the way COM access works in FoxPro, but provides not compile time type checking or Intellisense.*

What you need for this Article's Examples

- Visual FoxPro 9.0
- Visual Studio 2010 or 2012
- .NET 4.0 installed
- ASP.NET MVC 4.0
- IIS and ASP.NET Enabled on your machine (see top part here)

Creating your first COM component for ASP.NET

The first step is to create a COM object in Visual FoxPro and expose it as an MTDLL (multi-threaded STA) component. Although not really multi-threaded, STA allows COM to operate multiple instances of VFP components simultaneously providing a *simulation* of multi-threading for FoxPro COM components in multi-threaded environments like IIS and ASP.NET.

All the samples discussed are available in the download sample files which can be found at:

http://west-wind.com/files/Conferences/FoxProAspNetRevisited.zip

The code shown in Listing 1 is a very simple COM server that has two methods: The obligatory overly simple Helloworld() method and an Add() method calling a FoxPro server and returning some data.

FoxPro - A simple FoxPro Sample COM Server for use in ASP.NET

```
*************************************************************
DEFINE CLASS FirstServer AS Session OLEPUBLIC
*************************************************************

*** Some properties to access via COM
FirstName = "Rick"
Entered = {}


************************************************************************
*  HelloWorld
****************************************
FUNCTION HelloWorld(lcName as String) as string
IF EMPTY(lcName)
   lcName = "unknown"
ENDIF

RETURN "Hello World, " + lcName + ". Time is: " + TRANSFORM(DATETIME()) + "."
ENDFUNC
* HelloWorld

************************************************************************
*  Add
****************************************
FUNCTION Add(lnNumber as Currency, lnNumber2 as Currency) as Currency
RETURN lnNumber + lnNumber2
ENDFUNC
* Add

ENDDEFINE
```

Building the COM Server

To build this FoxPro class into a COM Server, create a new project called *foxaspnet* and add this class to it as the one and only source file. You can then build the project from the Project manager's build option by choosing the Multi-threaded DLL option or alternately using the following command from the Command Window:

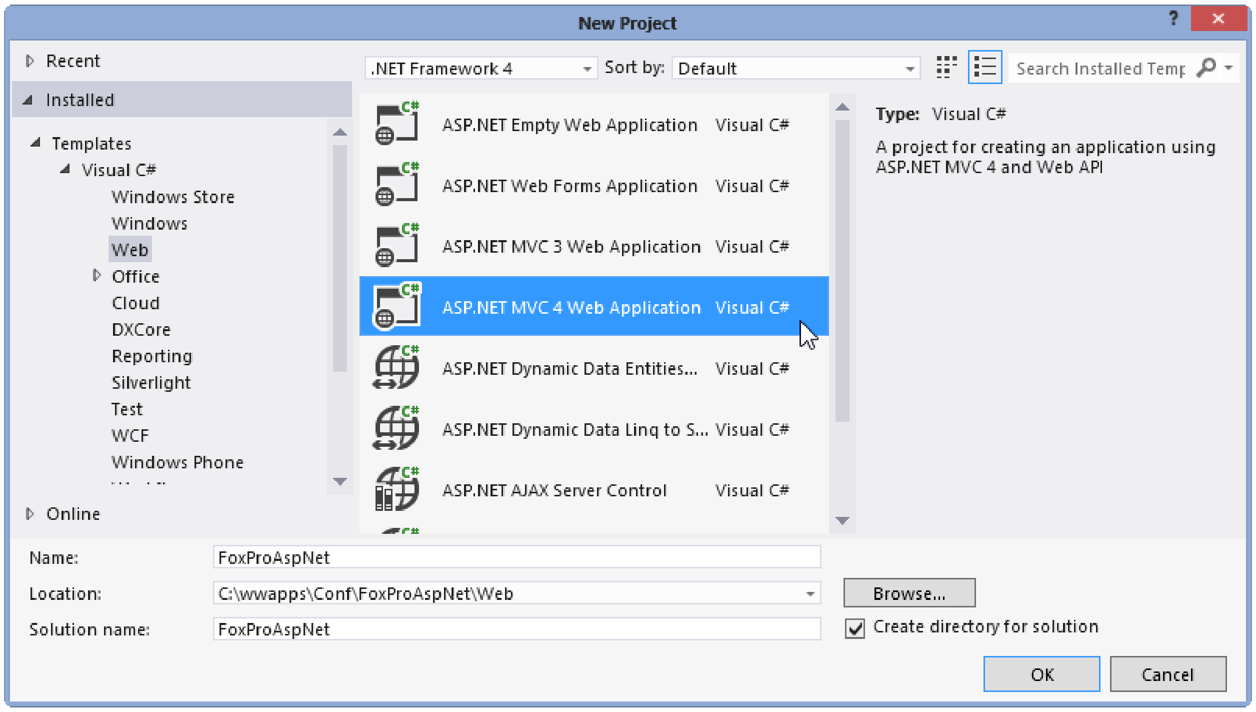
```
BUILD MTDLL foxaspnet FROM foxaspnet recompile
```

If your project compiles you should now have a COM object that is callable from .NET. Before you try it in .NET though you should test the server in Visual FoxPro to make sure it works. Try this from the FoxPro Command Prompt:

```
o = CREATEOBJECT("foxaspnet.firstFoxServer ")
? o.HelloWorld("Phoenix")
? o.Add(10,20)
o.Entered = DateTime()
o.FirstName = "Rick"
```

Create a new Visual Studio Project
Next let's create a new Visual Studio 2012 Web Project. This will also work with Visual FoxPro 2010 (just use MVC 3 instead of MVC 4).

- Start Visual Studio
- Select New Project
- Create a new Visual C# | Web | MVC 4 Web Project called FoxProAspNet
- Choose Empty Project type
  (or Basic if you want to have a bunch of project ready stuff imported – usually I end up removing most of this so it's easier to start with Empty and I'll end up adding a few things to this project that supercede the default stuff).


New Project
Recent
Installed
Templates
Visual C#
Windows Store
Windows
Web
Office
Cloud
DXCore
Reporting
Silverlight
Test
WCF
Windows Phone
Online
.NET Framework 4
Sort by: Default
Search Installed Temp
ASP.NET Empty Web Application Visual C#
ASP.NET Web Forms Application Visual C#
ASP.NET MVC 3 Web Application Visual C#
ASP.NET MVC 4 Web Application Visual C#
ASP.NET Dynamic Data Entities... Visual C#
ASP.NET Dynamic Data Linq to S... Visual C#
ASP.NET AJAX Server Control Visual C#
Browse...
OK
Cancel

Type: Visual C#

A project for creating an application using ASP.NET MVC 4 and Web API

Name: FoxProAspNet
Location: C:\wwapps\Conf\FoxProAspNet\Web
Solution name: FoxProAspNet
☑ Create directory for solution

Figure 3 – Creating a new ASP.NET MVC Project. Choose the Empty or Basic template.

When you're done you have a project that looks like this:

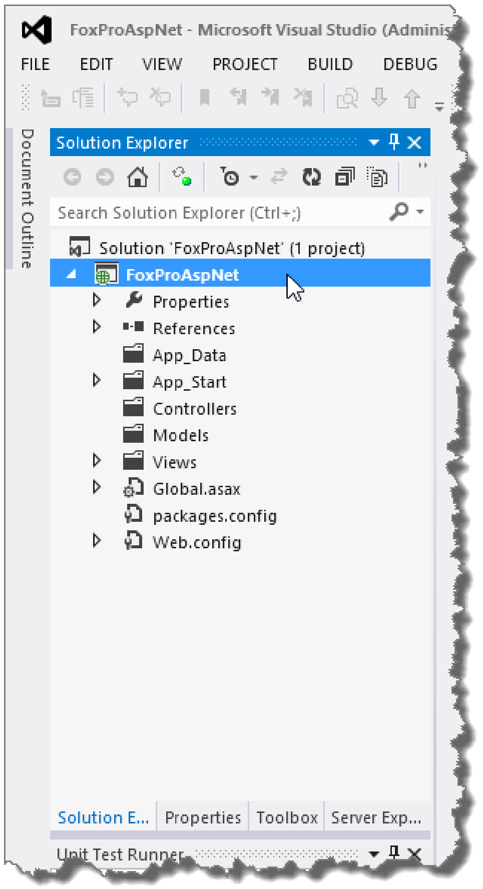


Figure 4 – The new ASP.NET MVC project once created.

Although I created an ASP.NET MVC project you can actually add any kind of ASP.NET item to this project including WebForms, Standalone Razor WebPage (.cshtml files), Web API controllers and ASMX and WCF Web Services.

In order to make our demos look nicer I'm going to add some CSS and some scripts into the project into top level css and scripts folder – you can check this out in the sample code provided – so the examples use some custom styling that is provided in these added files.

HelloWorld MVC

ASP.NET MVC has gotten to be the most popular ASP.NET technology for building HTML applications. It's based on the popular Model View Control (MVC) design pattern, where the **M**odel describes your data represented as objects, the **V**iew describes your display template (Razor .cshtml/.vbhtml Views) and the **C**ontroller which handles the unifying logic required to prepare the model and then pass it to the view for rendering. Essentially an MVC request consists of a controller method, that manipulates a model and sets its values, and then passes the model to View for display. The view then uses HTML templating to display the model data.

Controllers can also be 'headless' in ASP.NET MVC: They can run without a view and instead directly return data. For the simplest HelloWorld request we can write with ASP.NET MVC let's create a new controller and a HelloWorld method.

- Go the Project Root in Solution Explorer
- Right click and select Add New Item
- Select Add Controller
- Name the Controller FirstFoxController

A new class called FirstFoxController is created for you. I'm going to add a 'headless' HelloWorld method to the controller which then looks like this:

C# - Your first ASP.NET MVC Controller

```
public class FirstFoxController : Controller
{

    public ActionResult Index()
    {
        return View();
    }

    public string HelloWorld(string name)
    {
        dynamic fox = ComHelper.CreateObject("foxaspnet.FirstServer");
        return fox.HelloWorld(name);
    }
}
```

This controller method doesn't use a view but simply returns a string. To call this endpoint with MVC you can use the following URL in IIS:

http://localhost/foxProAspNet/FirstFox/HelloWorld?name=Rick

or if you didn't set up IIS and used IIS Express or the Visual Studio Web Server:
http://localhost:16706/FirstFox/HelloWorld?name=Rick

This produces very plain output that simply displays the data returned from our COM server which is:

Hello World, Rick. Time is: 09/17/12 05:20:41 PM.

How exciting ☺. But it demonstrates the basics of using the .NET 4 dynamic language features and creating a COM object. This code creates an instance of the COM object using late binding and then calls the Hello World method on the server.

Note the ComHelper.CreateObject() method used to create the COM instance since C# doesn't have a 'CreateObject' method natively. We'll be using this function a lot.

Understanding MVC Routing

In the previous request the URL to find our HelloWorld method is:

[http://localhost/foxProAspNet/**FirstFox/HelloWorld**?**name**=Rick](http://localhost/foxProAspNet/FirstFox/HelloWorld?name=Rick)

which maps to:

- FirstFoxController Class
- HelloWorld Method
- name Parameter

So how does MVC know how to find this URL? MVC uses a mechanism called routing and it implements a default route that maps routes to controllers and methods by URL path segments. The default route is defined as ASP.NET MVC's configuration (in App_Start\RegisterRoutes.cs) and it looks like this:

```
routes.MapRoute(
       name: "Default",
       url: "{controller}/{action}/{id}",
       defaults: new { controller = "Home", action = "Index",
                       id = UrlParameter.Optional }
);
```

This route says that the first parameter is the name of the controller (FirstFox pointing to FirstFox*Controller*), the action (the HelloWorld method) and an optional id value that we're not specifying here. The name parameter in the HelloWorld(string name) method is mapped from the query string, but it can also come from POST parameters. More on model binding later. Note that this is automatically set up for you, so this behavior is automatic for MVC projects. You can create custom routes using the same syntax as above where you map parameters by name in the {parametername} brackets to allow unique resource URLs.

Adding a View

Our first view isn't terribly useful for HTML generation although it can be useful for returning raw data. Let's add a view so we can display some HTML.

Views in MVC are created in the Views folder and you create Views in folders that match the controller names. So to create a HelloWorld view we first create a folder called *FirstFox* and then add the view there. To do this:

Add the folder:

- Find the Views folder in the project
- Right click Add | New Folder
- Name it FirstFox (to match the controller name)

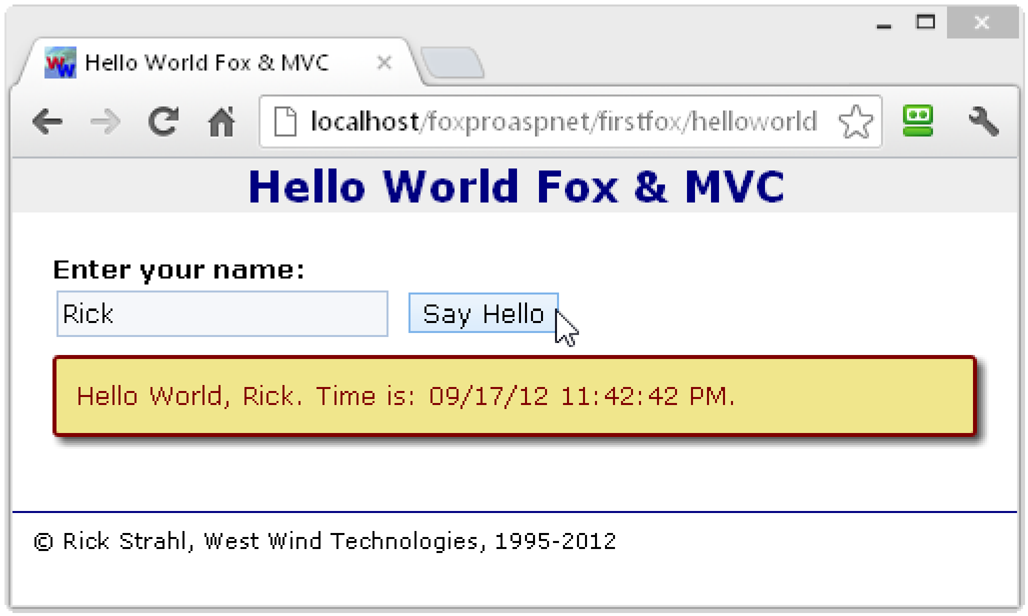
Create the View

- Right click on the Views/FirstFox folder
- Select View

- Name it HelloWorld

An ASP.NET MVC view is an HTML template that can contain markup code using C# (.cshtml) or VB (.vbhtml) code. The template language used is called Razor and is based on an inline language parsing engine which can mix code and HTML with minimal markup ‘noise’.

Let’s create a more interesting and interactive HTML view of our HelloWorld method and create a form that interactively asks for input like this:



Hello World Fox & MVC

Enter your name:

Rick

Hello World, Rick. Time is: 09/17/12 11:42:42 PM.

© Rick Strahl, West Wind Technologies, 1995-2012

Figure 5 – The interactive HelloWorld page

The Razor template HelloWorld.cshtml code to make this happen looks like this:

C# Razor – Our first Razor View for interactive HelloWorld

```
@{
    ViewBag.Title = "HelloWorld";
    Layout = null;
}
<html>
    <head>
        <title>Hello World Fox & MVC </title>
        <link href="~/Css/Reset.css" rel="stylesheet" />
        <link href="~/Css/Standard.css" rel="stylesheet" />
    </head>
<body>

<h1>Hello World Fox & MVC</h1>

@using( Html.BeginForm()) {
<div class="containercontent">
    <label>Enter your name:</label>
    <div class="inputfield">
        @Html.TextBox("name", ViewBag.Name as string)
        <input type="submit" id="btnSubmit" name="btnSubmit" value="Say
Hello" />
    </div>

    @if(!string.IsNullOrEmpty(ViewBag.Message as string)) {
    <div class="errordisplay">
        @ViewBag.Message
    </div>
    }
</div>
}

    <footer class="footer">
    &copy; Rick Strahl, West Wind Technologies, 1995-@DateTime.Now.Year
    </footer>
</body>
</html>
```

As you can see the template mixes HTML and code in a semi fluent fashion.

@ C# Expressions and CodeBlocks
Razor uses @ to embed C# expressions into the page and **@{ }** to embed C# code blocks. C# code and HTML blocks are nearly seamlessly integrated with each other – Razor understands C# (and VB) syntax and so can detect codeblocks. The result is a fairly lean template language that's a bit cleaner than the classic ASP.NET WebForms or ASP classic <% %> syntax.

Notice that there are several **@{ }** blocks – the **@Html.Form()** block that creates a form element (you can also create this manually if you choose, but it's easier and more self-maintaining this way), the **@Html.TextBox()** helper that creates an input element including automatic databinding of the name to the textbox. Finally there's a conditional **@if** block on the bottom that is used to determine whether or not the Message needs to be displayed.

The ViewBag – the poor Man's Model
In this example, we don't have an explicit model, but here I use the ViewBag object to pass 'model' data from the Controller to the View. A View knows nothing of the Controller, so anything you want to display in the view you have to pass to the View. What's passed is the the **M** in MVC: You pass a model (or more commonly a ViewModel).

ViewBag is an expandable dynamic object on which you can simply create properties simply by referencing them. Both the Controller and View have ViewBag properties that reference this same object, so it's an easy way to share data when you don't want to pass an explicit model to the View.

In this view I have two values I'm passing via the ViewBag object: Message and Name properties. The name passes the name to display in the textbox and the message passes the message. Note that I'm not calling the COM object in my View – while you can easily pass the COM object to the View it's better to set the value on ViewBag in the controller and then use the ViewBag/Model value in the view.

Creating the Controller to drive the View

The Controller code is responsible for setting up the 'model' which in this case is the ViewBag object. So let's change our HelloWorld controller to set up the ViewBag.

Here's the new HelloWorld method that handles both showing the initial form and submitting the name for display from a single method:

C# - Rewriting the HelloWorld Controller to be interactive

```
public ActionResult HelloWorld(string name = null)
{
   dynamic Fox = ComHelper.CreateObject("foxaspnet.FirstServer");

   // assign a value to the ViewBag Container
    ViewBag.Name = name;

    if (!string.IsNullOrEmpty(name))
        ViewBag.Message = Fox.HelloWorld(name);
    else
        ViewBag.Message = null;

    return this.View();
}
```

The method is pretty simple yet handles both the initial display and the POST operation that accepts the user name from the HTML form.

The *name* parameter is automatically filled either from the query string as we did earlier or from the POST buffer when the user submits the page via Say Hello button. ASP.NET uses ModelBinding to automatically map parameters from the form buffer or query string and automatically performs type conversions for non-string types. For more complex scenarios model binding can automatically bind properties of an object to form or query string variables. We'll look at that later when we capture more user input.

This code creates our FoxPro COM object then checks whether a name was passed. If it is we create and set the Message property on the ViewBag object. If no name was passed the message is set to null so the message display box doesn't show (@if block in HTML).

Finally the *this.View()* method is called, which tells MVC to load the convention of loading the View that has the same name as the controller method (Helloworld). It loads and executes the /Views/FirstFox/Helloworld.cshtml view page and passes the ViewBag object to the view. I'm using the default View() method here, but it has many overloads that allow you specify which View to display and to pass an explicit model object to it. We'll do this in later examples. For now just realize that the defaults follow a convention of using the same ViewName as the Controller method.

Adding a new Method to our COM Server

Phew! The last section included a lot of MVC specific behavior that you need to know. It might seem a little overwhelming at first, but you've just gotten a crash course in MVC operational behavior and you now have most of what you need to do 90% of your MVC development.

So, let's apply what we learned by creating another request with something a little more interesting. Let's add another method to our COM server that retrieves a stock quote and returns stock info as an object. I'll add a method to the FirstServer COM class that retrieves a stock quote:

FoxPro – A COM Server method that returns a StockQuote as an object

```
************************************************************************
*  GetStockQuote
****************************************
FUNCTION GetStockQuote(lcSymbol as string) as Object

IF EMPTY(lcSymbol)
      lcSymbol = ""
ENDIF

loServer = CREATEOBJECT("YahooStockServer")
loQuote = loServer.GetStockQuote(lcSymbol)
RETURN loQuote
ENDFUNC
*   GetStockQuote
```

The stock quote lookup logic is part of the samples and can be found in YahooStockServer.prg – it uses the Yahoo online live quote service to retrieve stock qutoes via HTTP. Yahoo provides stock quotes based on simple URLs in CSV format. The code in the server makes HTTP calls to retrieve this data and parses the results into a StockQuote object that is returned from the GetStockQuote() method.

Once you've added this method to the server, you should now re-compile the server with:

```
BUILD MTDLL foxaspnet FROM foxaspnet
```

Ooops! If you do that now, you'll likely get an error:

File access is denied c:\wwapps\conf\foxproaspnet\foxpro\foxaspnet.dll.

The file is locked! The problem is that the COM object is loaded inside of IIS and so needs to be unloaded in order to properly recompile. Effectively you need to shut down IIS (or more specifically the IIS Application that hosts your COM object).

The easiest way to do this is by doing running the **IISReset** utility (installed with the IIS Management tools) or you can alternately use the IIS Management Console to Recyle the IIS server.

I have an Intellisense shortcut setup for IIS that does:

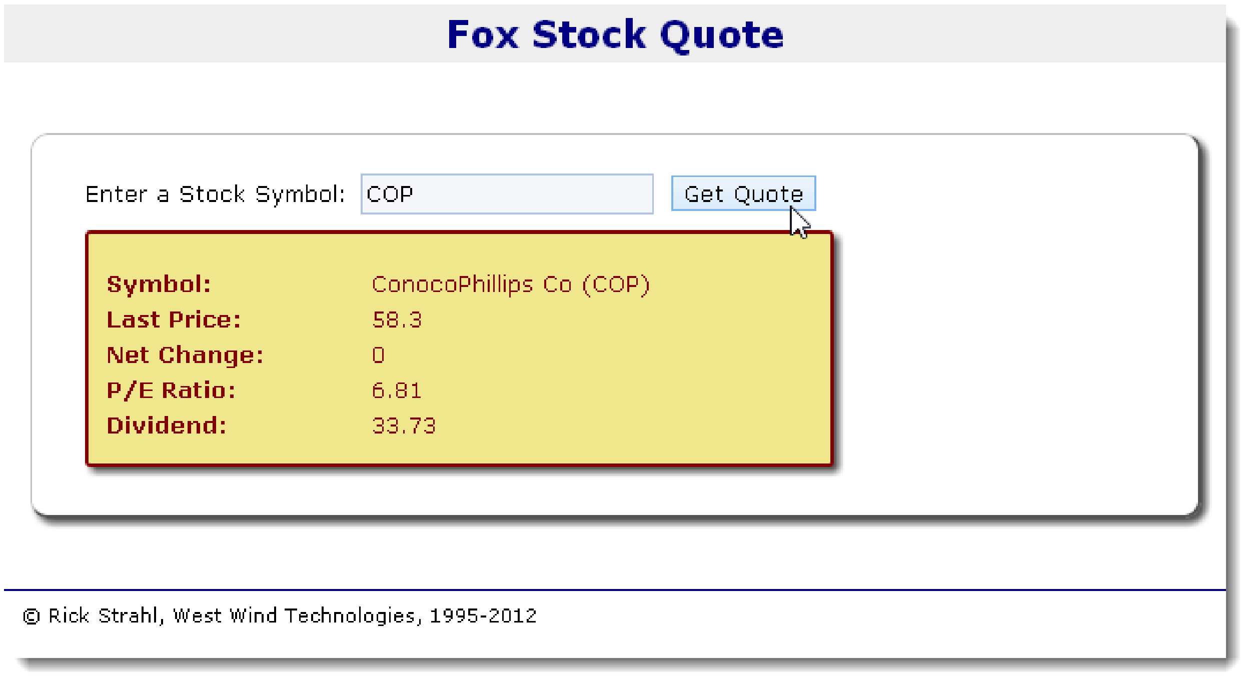
```
RUN /n4 IISRESET
```

> **IISRESET requires true Admin Priviliges**
> Note that you need to be a full administrator to do this. If UAC is on you have to 'Run As Administrator' when you run IISRESET.exe. If you run with UAC it's a good idea to create a batch file that executes IIS Reset, and pin it to your startmenu, then set the properties to Run As Administrator always.

Once you've unloaded the DLL you can now recompile it. Once recompiled try the COM object from the FoxPro command line:

```
loFox = CREATEOBJECT("foxaspnet.firstserver")
loQuote = loFox.GetStockQuote("msft")
? loQuote.Company
? loQuote.Price
? loQuote.LastPrice
? loQuote.NetChange
? loQuote.PeRatio
```

Ok, now that it works let's hook this up to our .NET Controller to produce this Web UI:



Symbol:	ConocoPhillips Co (COP)
Last Price:	58.3
Net Change:	0
P/E Ratio:	6.81
Dividend:	33.73

© Rick Strahl, West Wind Technologies, 1995-2012

Figure 6 – The Fox Stock Quote Example retrieves live stock quotes and stock info

Let's start with the Controller this time. The controller code is super simple:

C# - The StockQuote Controller simply forwards the FoxPro COM call result

```
public ActionResult StockQuote(string symbol)
```

```
{
    dynamic Fox = ComHelper.CreateObject("foxaspnet.FirstServer");
    dynamic quote = Fox.GetStockQuote(symbol);
    return View(quote);
}
```

The code here simply calls the GetStockQuote() method on our Fox COM object which returns a quote object. This quote object is then used as the model to the view. Note I don't use ViewBag here, but rather pass the quote object as the model to the View. The quote is a FoxPro COM object that holds only the quote result data which in typical light-weight model fashion is desirable for the MVC pattern.

The View then uses the Model internally to display the results and handle the symbol input. Here's the View:

C# Razor – The StockQuote View uses the Quote object model to display stock info

```
@{
    ViewBag.Title = "StockQuote";
}
@section HtmlHeader{
    <style>
    .fieldwrapper
    {
        clear: both;
        margin: 10px 0;
        min-height: 10px;
    }
    </style>
}
<h1>Fox Stock Quote</h1>

<div class="contentcontainer" style="width: 600px;margin: 40px auto;">

    @using(Html.BeginForm()) {

    <span>Enter a Stock Symbol:</span>
    @Html.TextBox("symbol")
    <input type="submit" id="btnSubmit" name="btnSubmit" value="Get Quote" />


    <div id="StockDisplay" class="errordisplay"  style="width: 400px;">

        <div class="fieldwrapper">
            <div class="label-left">Symbol:</div>
            <div class="leftalign">
                @Model.Company (@Model.Symbol)
            </div>
        </div>

        <div class="fieldwrapper">
            <div class="label-left">Last Price:</div>
            <div class="leftalign">
                @Model.LastPrice
            </div>
        </div>

        <div class="fieldwrapper">
            <div class="label-left">Net Change:</div>
            <div class="leftalign">
                @Model.NetChange
            </div>
        </div>
        <div class="fieldwrapper">
            <div class="label-left">P/E Ratio:</div>
            <div class="leftalign">
                @Model.PeRatio
            </div>
        </div>
        <div class="fieldwrapper">
            <div class="label-left">Dividend:</div>
            <div class="leftalign">
                @Model.DividendYield
            </div>
        </div>

        <div  class="clearfix"></div>
    </div>
    }

</div>
```

The view is pretty simple in that it merely echoes back values from the Model and posts back the symbol value to the server. When the button is pressed the symbol input box value is sent to the controller and mapped to the symbol parameter of the StockQuote method. MVC automatically maps POST variables to matching parameters.

This View takes advantage of a few additional features of MVC.

Dynamic Model Usage
In this example I'm not using the ViewBag object, but rather use the dynamic Model property. When I call *View(quote)* I pass in the quote as the Model for this page and the quote then becomes available as the *Model* property in the page.

The display of the quote data is then done by using simple @Model.Property expressions rendered into the page accessing our FoxPro data directly. The Quote object is in fact a FoxPro COM object the properties of which we are accessing here.

Layout Pages
First off notice that this page is not a complete HTML page – rather it merely contains the 'content' of the page. It's relying on a Layout template to handle displaying the base page layout that contains the HTML header and a few other things. By default MVC creates a /Views/Shared/_Layout.cshtml page and links that to all views by default via the Layout property (defined in /Views/Shared/_ViewStart.cshtml page.)

The _Layout.cshmtl page contains a base frame for pages:

C# Razor – A master Layout page is a master template into which views can load

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link href="~/Css/reset.css" rel="stylesheet" type="text/css" />
        <link href="~/Css/standard.css" rel="stylesheet" type="text/css" />

        @RenderSection("HtmlHeaderTop",false)
        <title>@ViewBag.Title</title>
        @RenderSection("HtmlHeader",false)
        <meta charset="utf-8" />
        <link href="~/favicon.ico" rel="shortcut icon" type="image/x-icon" />
        <script src="~/Scripts/jquery.min.js"></script>
    </head>
    <body>
        @RenderBody()
        <footer class="footer">
        &copy; Rick Strahl, West Wind Technologies, 1995-@DateTime.Now.Year
       </footer>
    </body>
</html>
```

The key item in this master page is the *@RenderBody()* method call which renders the actual page – the StockQuote.cshtml page in this case. I also have two option *@RenderSection()* commands in the layout template that allows rendering header information into the page. The StockQuote page uses the @section HtmlHeader {} to add custom styles to the header of the page for example.

The layout template can be explicitly specified in the View header via:

```
@{
       Layout = "~/Views/Shared/_WebLogLayout.cshtml";
}
```

By default, ~/Views/Shared/_Layout.cshtml is used for layout as defined in /Views/Shared/_ViewStart.cshtml which acts as a 'startup view' that is fired before any of the other templates are loaded. IOW, every page renders with _Layout.cshtml by default, unless you specify otherwise.

If you want to render a page without the default layout if one is set in _ViewStart, just set the Layout to null:

```
@{
    Layout = null;
}
```

Now the page only renders what you put into the specific view you are working on.

Adding Data Access and sharing Data with .NET

Next let’s look at passing data from FoxPro to .NET. Up to now we’ve passed values and objects around which is straight forward. FoxPro data in Cursors/Tables however can’t be passed over COM.

Since you can’t pass cursors directly, here are a few other options to return data:

- **Collection**
- XML
- HTML

I covered XML and HTML results in the old article. Another good way to return data to .NET is to convert your data into a collection. Prior to dynamics this didn’t work very well because typelibrary exports tended to mangle the collection object passed back to .NET. With dynamic however a FoxPro Collection object is easily accessed in .NET.

To do this I’ll add another method to my server. In this example I’m going to retrieve some WebLog posts from my Weblog which sits in data from the \weblog folder in the samples. The data is in FoxPro DBF files.

In a FoxPro app you might write a simple method that returns some data like this:

FoxPro – Retrieving some FoxPro data as part of a COM object

```
************************************************************************
*  GetRecentBlogEntries
****************************************
FUNCTION GetRecentBlogEntries()

SELECT TOP 10 * FROM Blog_Entries ;
  INTO CURSOR TEntries ;
  ORDER BY Entered DESC

RETURN _TALLY
ENDFUNC
*   GetRecentBlogEntries
```

This code has a few problems – not the least of which is that we can’t return the cursor to .NET. But first let’s see what happens if we try to run the code as is.

Unload IIS and recompile your DLL, then create a new Controller method in FirstServerController.cs.

C# - Reading the query result in a Controller method

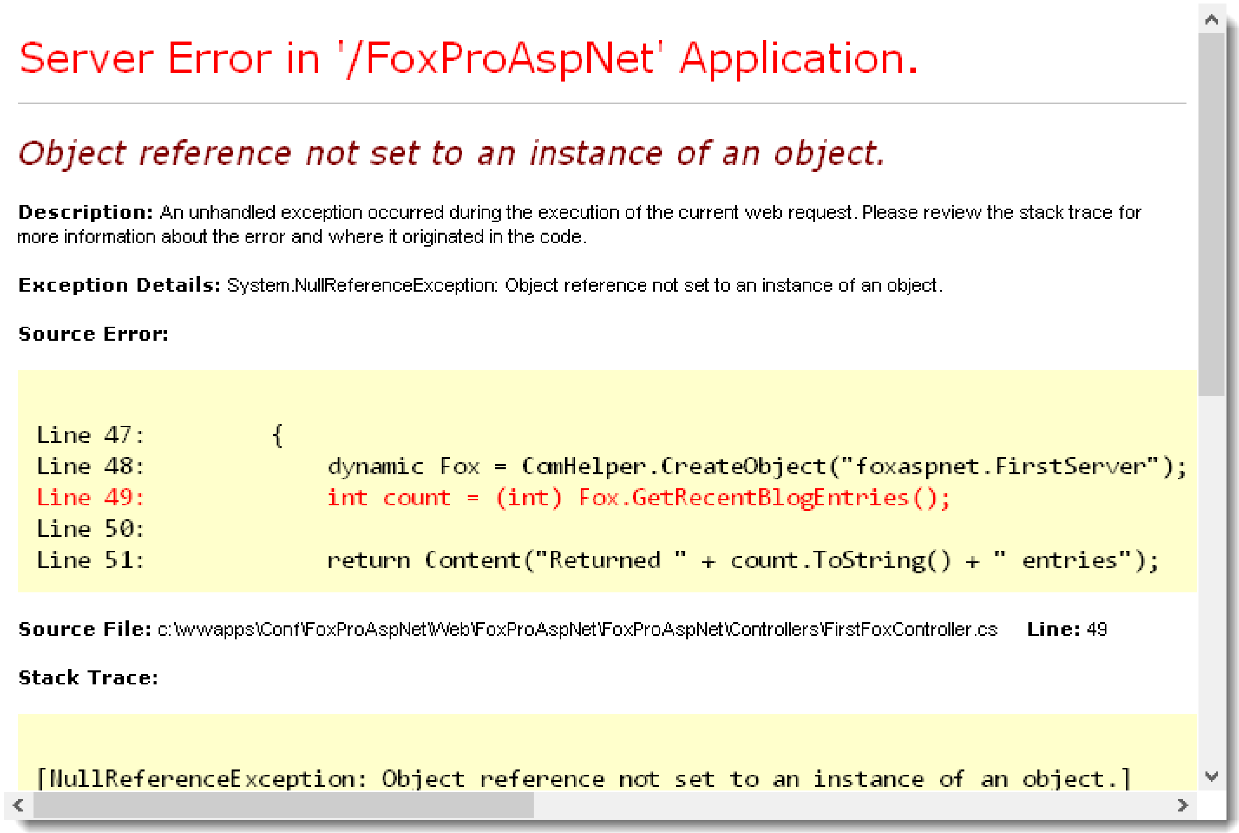
```
public ActionResult RecentBlogEntries()
{
    dynamic Fox = ComHelper.CreateObject("foxaspnet.FirstServer");
    int count = (int) Fox.GetRecentBlogEntries();
    return Content("Returned " + count.ToString() + " entries");
}
```

All this method does – for now – is simply echo back the _Tally value. Compile (F6) in Visual Studio and bring up your browser and navigate to:

http://localhost/FoxProAspNet/FirstFox/RecentBlogEntries

Error, Error on the Wall!

This request fails and we've now hit our first FoxPro error in a COM object. When the smoke clears you end up with an ASP.NET Yellow Screen of Death error page like this:



Server Error in '/FoxProAspNet' Application.

Object reference not set to an instance of an object.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.NullReferenceException: Object reference not set to an instance of an object.

Source Error:

```
Line 47:         {
Line 48:             dynamic Fox = ComHelper.CreateObject("foxaspnet.FirstServer");
Line 49:             int count = (int) Fox.GetRecentBlogEntries();
Line 50:
Line 51:             return Content("Returned " + count.ToString() + " entries");
```

Source File: c:\wwapps\Conf\FoxProAspNet\Web\FoxProAspNet\FoxProAspNet\Controllers\FirstFoxController.cs **Line:** 49

Stack Trace:

```
[NullReferenceException: Object reference not set to an instance of an object.]
```

Figure 7 – A COM Error in our Fox object gives a vague error message with *dynamic*.

It shows us the error that occurs in .NET but unfortunately no further error information in Visual FoxPro. Rats... Unfortunately this is one of the downsides of the new *dynamic* functionality – when an error occurs in the COM object we just get a generic error returned.

If you want to find out what REALLY happened you have to resort to Reflection to get more information unfortunately. The ComHelper class includes CallMethod() that does just that.

FoxPro – COM Helper to invoke a method via Reflection to provide error info

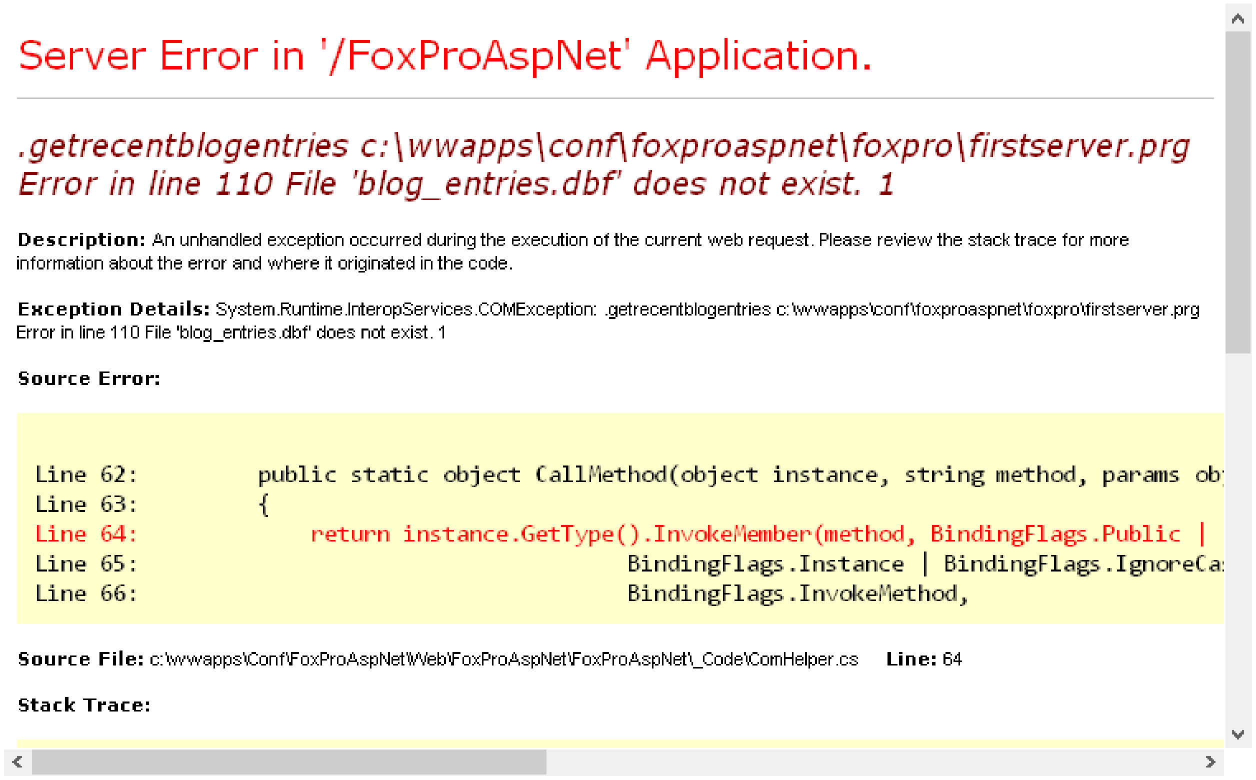
```
public static object CallMethod(object instance, string method,
                                params object[] parms)
{
    return instance.GetType().InvokeMember(method, BindingFlags.Public |
                            BindingFlags.Instance | BindingFlags.IgnoreCase|
                            BindingFlags.InvokeMethod,
                            null, instance, parms);
}
```

To use it, replace the dynamic call to GetRecentBlogEntries() line temporarily with:

```
int count = (int)ComHelper.CallMethod(Fox, "GetRecentBlogEntries");
```

Call method accepts an object instance, the name of the method to call and any number of optional parameters to pass.

Compile the code and re-run. We still get an error, but now we should see a bit more error information on the error page, with error info from the FoxPro COM object bubbling up to the Web page.



Server Error in '/FoxProAspNet' Application.

.getrecentblogentries c:\wwapps\conf\foxproaspnet\foxpro\firstserver.prg Error in line 110 File 'blog_entries.dbf' does not exist. 1

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.Runtime.InteropServices.COMException: .getrecentblogentries c:\wwapps\conf\foxproaspnet\foxpro\firstserver.prg Error in line 110 File 'blog_entries.dbf' does not exist. 1

Source Error:

```
Line 62:         public static object CallMethod(object instance, string method, params ob
Line 63:         {
Line 64:             return instance.GetType().InvokeMember(method, BindingFlags.Public |
Line 65:                                     BindingFlags.Instance | BindingFlags.IgnoreCa
Line 66:                                     BindingFlags.InvokeMethod,
```

Source File: c:\wwapps\Conf\FoxProAspNet\Web\FoxProAspNet\FoxProAspNet_Code\ComHelper.cs **Line:** 64

Stack Trace:

Figure 8 – When using Reflection to invoke the method with an error you get a lot more error information. The full FoxPro error message including line of code causing the error is returned.

Aha! Now we're getting somewhere. The error occurs because the data file I'm looking for is not found which makes perfect sense. Our COM Server runs in the context of the hosting application – IIS in this case, and so it has no idea how to find my Blog_Entries table in the query. I need to make sure that the environment is properly set.

Setting the FoxPro Environment for COM

When you create a top level COM object, it's important to set the environment. You need to set the Application's path so data can be found, set any environment settings to ensure that your server works properly. Basically every time a new COM object is instantiated it's treated as its own new instance with a new FoxPro environment.

The best way to do this is to use a base class that performs a few common tasks. I use an AspBase class to handle this for me and I inherit any top level COM objects off this class.

FoxPro – AspBase COM base class provides common environment configuration

```
*************************************************************
DEFINE CLASS AspBase AS Session
*************************************************************
***    Author: Rick Strahl
***            (c) West Wind Technologies, 1999
***   Contact: (541) 386-2087  / rstrahl@west-wind.com
***  Modified: 12/08/98
***  Function: ASP Base class that provides basic
***            requirements for safe execution of a COM
***            component in ASP
*************************************************************

*** Hide all VFP members except Class
PROTECTED AddObject, AddProperty, Destroy, Error, NewObject, ReadExpression,
Readmethod, RemoveObject, ResetToDefault, SaveAsClass, WriteExpression,
WriteMethod, BaseClass, ClassLibrary, Comment, Controls, ControlCount,
Height, Width, HelpContextId, Objects, Parent, ParentClass, Picture,
ShowWhatsThis, WhatsThisHelpId

*** Base Application Path
ApplicationPath = ""

*** .T. if an error occurs during call
IsError = .F.

*** Error message if an error occurs
ErrorMessage = ""

*** Name of the Visual FoxPro Class - same as Class but Class is hidden
cClass = ""

ComHelper = null

************************************************************************
* WebTools :: Init
*********************************
***  Function: Set the server's environment. IMPORTANT!
```

```
************************************************************************
FUNCTION Init

*** Expose Class publically
this.cClass = this.Class

*** Make all required environment settings here
*** KEEP IT SIMPLE: Remember your object is created
***                 on EVERY ASP page hit!
IF Application.StartMode > 1  && Interactive or VFP IDE COM
      SET RESOURCE OFF  && best to do in compiled-in CONFIG.FPW
ENDIF

*** SET YOUR ENVIRONMENT HERE (for all)
*** or override in your subclassed INIT()
SET EXCLUSIVE OFF
SET DELETED ON
SET EXACT OFF
SET SAFETY OFF

*** Retrieve the DLL location and grab just the path to save!
THIS.ApplicationPath = GetApplicationPath() && ADDBS(

*** Com Helper that helps with Conversions of cursors
THIS.ComHelper = CREATEOBJECT("ComHelper")

*** Add the startup path to the path list!
*** Add any additional relative paths as required
SET PATH TO ( THIS.ApplicationPath )

ENDFUNC


************************************************************************
*  SetError
****************************************
PROTECTED FUNCTION SetError(lcMessage, lcMethod, lnLine)

IF ISNULL(lcMessage) OR EMPTY(lcMessage)
   this.ErrorMessage = ""
   this.IsError = .F.
   RETURN
ENDIF

THIS.IsError = .T.
THIS.ErrorMessage= lcMessage

ENDFUNC
*   SetError

ENDDEFINE
* AspBase

************************************************************************
FUNCTION GetApplicationPath
*********************************
```

```
***  Function: Returns the FoxPro start path
***            of the *APPLICATION*
***            under all startmodes supported by VFP.
***            Returns the path of the starting EXE,
***            DLL, APP, PRG/FXP
***    Return: Path as a string with trailing "\"
*************************************************************************

DO CASE
   *** ServerName property for COM servers EXE/DLL/MTDLL
   CASE INLIST(Application.StartMode,2,3,5)
         lcPath = JustPath(Application.ServerName)

   *** Interactive
   CASE (Application.StartMode) = 0
         lcPath = SYS(5) + CURDIR()

  *** Standalone EXE or VFP Development
  OTHERWISE
       lcPath = JustPath(SYS(16,0))
       IF ATC("PROCEDURE",lcPath) > 0
         lcPath = SUBSTR(lcPath,RAT(":",lcPath)-1)
       ENDIF
ENDCASE

RETURN ADDBS(lcPath)
```

The most important task this class performs is to set the Application Path. Basically it finds out where the DLL invoked lives and automatically sets the path to this folder and also sets the *ApplicationPath* property to this path. This path is crucial to finding your data and accessing it in the COM object.

The class also sets up an error and error message property that you can use to store error information to pass back to the calling application.

The ComHelper Property

The AspBase class also has a *ComHelper* property. This object provides a bunch of helper functionality that is useful for converting data:

- CreateObject
- CursorToCollection
- CursorToXmlString
- XmlStringToCursor

We'll use CreateObject() to create regular FoxPro object and pass them to ASP.NET basically bypassing standard COM invocation. It's a great way to expose objects to COM without explicitly making them COM objects. CursorToCollection() allows us to turn a FoxPro cursor into a Collection object which can then be passed to .NET. It's an easy way to pass data to .NET.

Implementing AspBase

To use AspBase simply use it as a base class for any top level COM class. The class then inherits all the AspBase properties including the ApplicationPath, ErrorMessage and ComHelper properties. Let's use it to enhance the FoxFirstServer class.

FoxPro – Enhancing our FoxPro COM object by inheriting from the AspBase class

```
*************************************************************
DEFINE CLASS FirstServer AS AspBase OLEPUBLIC
*************************************************************

************************************************************************
*  Override the Init
****************************************
FUNCTION Init()

*** call the base method to do base configuration
DODEFAULT()

SET PATH TO THIS.ApplicationPath + "data\;" + ;
            THIS.ApplicationPath + "weblog\;"


ENDFUNC

*** Other methods go here…

ENDDEFINE
```

Note that I simply override the Init() method and call back to the base behavior with DoDefault(). The base behavior sets the ApplicationPath and ComHelper properties so make sure this is called. After the call ApplicationPath is set to the path of the DLL which you can then use to add additional paths to point at your data or any other resources you might need.

This class inherits all the helpful properties like ApplicationPath and ComHelper etc. I'm also setting the path explicitly to the paths where I look for data – namely the data and weblog folders which is the key to making the previously failing GetRecentBlogEntries() method work. Specifically here I set the path to include the weblog\ folder which contains the blog_Entries table required for the GetRecentBlogEntries() method call.

Unload IIS, recompile the COM server now and then re-run the browser request for:

http://localhost/FoxProAspNet/FirstFox/RecentBlogEntries

and now you should get a result of:

Returned 10 entries

Sweet – the method now works and we can access the data files.

From Cursor to Object

But we have another issue now. The method worked and a cursor was created in FoxPro, but we can’t access this FoxPro cursor in .NET. The problem is that the cursor lives inside of the FoxPro dll and we can’t pass the cursor over COM.

What we can do however is convert the cursor into something that COM can use. So let’s change the controller method:

FoxPro – Using CursorToCollection to convert a Fox Cursor to an object

```
public ActionResult RecentBlogEntries()
{
    dynamic Fox = ComHelper.CreateObject("foxaspnet.FirstServer");
    Fox.GetRecentBlogEntries();

    dynamic entries = Fox.ComHelper.CursorToCollection("TEntries");

    return View(entries);
}
```

The code creates the COM object then calls the GetRecentBlogEntries() method to retrieve the cursor. In the business object this creates a cursor called TEntries which in FoxPro you could then use in your UI. Here in ASP.NET we can’t access the cursor, but it’s still open and alive in the COM object. So then I can use the *ComHelper.CursorToCollection()* method to specify the name of the cursor and turn it into a collection object that I can access in .NET.

CursorToCollection is fairly simple – it simply scans through a FoxPro cursor and creates a Collection of SCATTER NAME objects:

FoxPro – Implementing CursorToCollection is easy and fairly fast

```
FUNCTION CursorToCollection(lcCursor)
LOCAL loItem, loCol as Collection

loCol = CREATEOBJECT("Collection")
IF EMPTY(lcCursor)
  lcCursor = ALIAS()
ENDIF

SELECT (lcCursor)
SCAN
      SCATTER NAME loItem MEMO
      loCol.Add(loItem)
ENDSCAN

RETURN loCol
```

Since the collection is an object rather than a FoxPro cursor, this object can be passed over COM to .NET and we can pass this collection to the view.

C# Razor – Rendering the WebLog Entries Collection to HTML

The BlogPost Entry Collection becomes the Model for the View I want to display. The view loops through the entries and displays items. It lives in /Views/foxFirst/RecentBlogEntries.cshtml and looks like this:

```
@{
    ViewBag.Title = "Rick's FoxPro Blog";
}
@section HtmlHeader
{
    <style>
        .blogentry {
            padding: 10px;
            border-bottom: dashed 1px teal;
            min-height: 60px;
        }
        .blogtitle {
            font-weight: bold;
            font-size: 1.1em;
            color: Steelblue;
        }
        .blogabstract {
            font-size: 0.8em;
            margin: 5px 10px;
        }
        .blogabstract i{
            font-size: 1.2em;
            font-weight: bold;
        }
        .contentcontainer{
            margin: 40px auto;
            width: 700px;
        }
    </style>
}

<h1>@ViewBag.Title</h1>

<div class="contentcontainer">
    <div class="gridheader">Blog Entries</div>
    @{ for (int i = 1; i <= Model.Count; i++)
       {
           dynamic entry = Model.item(i);

            <div class="blogentry">
                <div class="blogtitle">
                    @entry.Title
                </div>
                <div class="blogabstract">
                    <i>@entry.Entered.ToString("MMM dd")</i> - @entry.Abstract
                </div>
            </div>
       }
    }
</div>
```

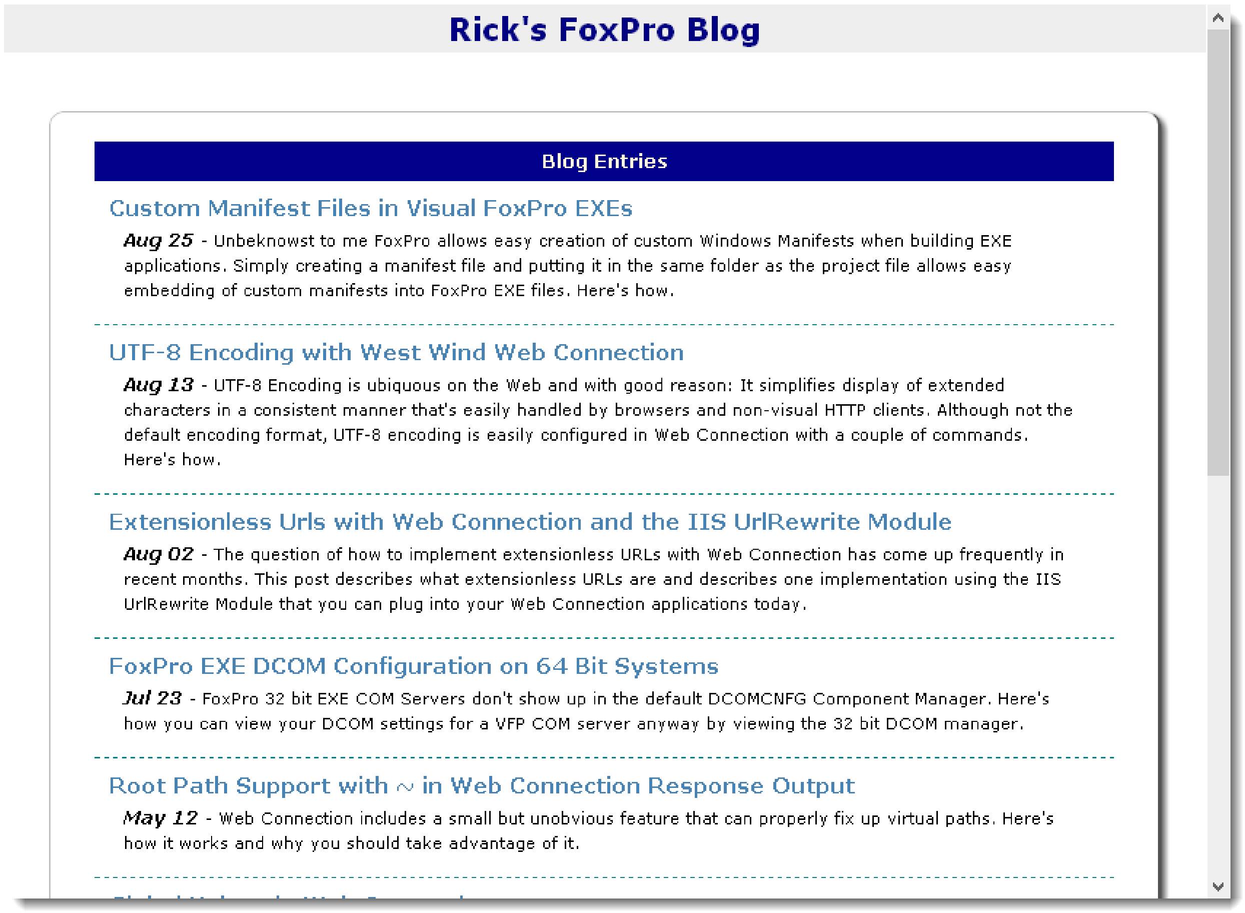
The key code is the *for* structure that loops through the collection items. Note that FoxPro collections don't show up as .NET collections so you can't use foreach(), but rather you have to iterate over the list with an index. Also note that the index is 1 based, not 0 based like most other .NET collections and arrays.

To access an individual Entry object you can use code like this:

```
dynamic entry = Model.Item(i);
```

which matches the FoxPro Collection structure. All that's left to do then is to render some of the data into the page using @Model.Title style syntax for each of the items inside of the for loop.

Here's what the rendered page looks like.



Rick's FoxPro Blog

Blog Entries

Custom Manifest Files in Visual FoxPro EXEs

Aug 25 - Unbeknowst to me FoxPro allows easy creation of custom Windows Manifests when building EXE applications. Simply creating a manifest file and putting it in the same folder as the project file allows easy embedding of custom manifests into FoxPro EXE files. Here's how.

UTF-8 Encoding with West Wind Web Connection

Aug 13 - UTF-8 Encoding is ubiquous on the Web and with good reason: It simplifies display of extended characters in a consistent manner that's easily handled by browsers and non-visual HTTP clients. Although not the default encoding format, UTF-8 encoding is easily configured in Web Connection with a couple of commands. Here's how.

Extensionless Urls with Web Connection and the IIS UrlRewrite Module

Aug 02 - The question of how to implement extensionless URLs with Web Connection has come up frequently in recent months. This post describes what extensionless URLs are and describes one implementation using the IIS UrlRewrite Module that you can plug into your Web Connection applications today.

FoxPro EXE DCOM Configuration on 64 Bit Systems

Jul 23 - FoxPro 32 bit EXE COM Servers don't show up in the default DCOMCNFG Component Manager. Here's how you can view your DCOM settings for a VFP COM server anyway by viewing the 32 bit DCOM manager.

Root Path Support with ~ in Web Connection Response Output

May 12 - Web Connection includes a small but unobvious feature that can properly fix up virtual paths. Here's how it works and why you should take advantage of it.

Figure 9 – The rendered Web Log Entries page is basic but functional

A WebLog Example

Let's continue on with another more realistic example using the WebLog theme I started with the last example. I'm going to create another FoxPro COM object that handles my WebLog related tasks.

I like to minimize the amount of COM objects I have to explicitly create in FoxPro. In this section I'll describe how you can dynamically create FoxPro objects without those objects having to be explicit COM objects which avoids the issues of setting up the environment or having to be based on the AspBase class which effectively would require changes to existing objects.

Rather I prefer to create one top level COM object, that can then create other FoxPro objects and pass them over COM to our .NET controller code. This way I can use existing business object classes without having to modify them in any way as they inherit the environment from the top level COM object.

In this example, I'm going to use a couple of business objects – blog_entry and blog_comment - rather than accessing data directly in a front end method. These business objects are based on my own wwBusiness class (part of West Wind Web Connection or the West Wind Client Tools), but any business object framework could be used instead. The samples include all the necessary files.

I'll start with my FoxPro class that is exposed as a COM object:

FoxPro – The Top Level WebLog COM Object – not much functionality here

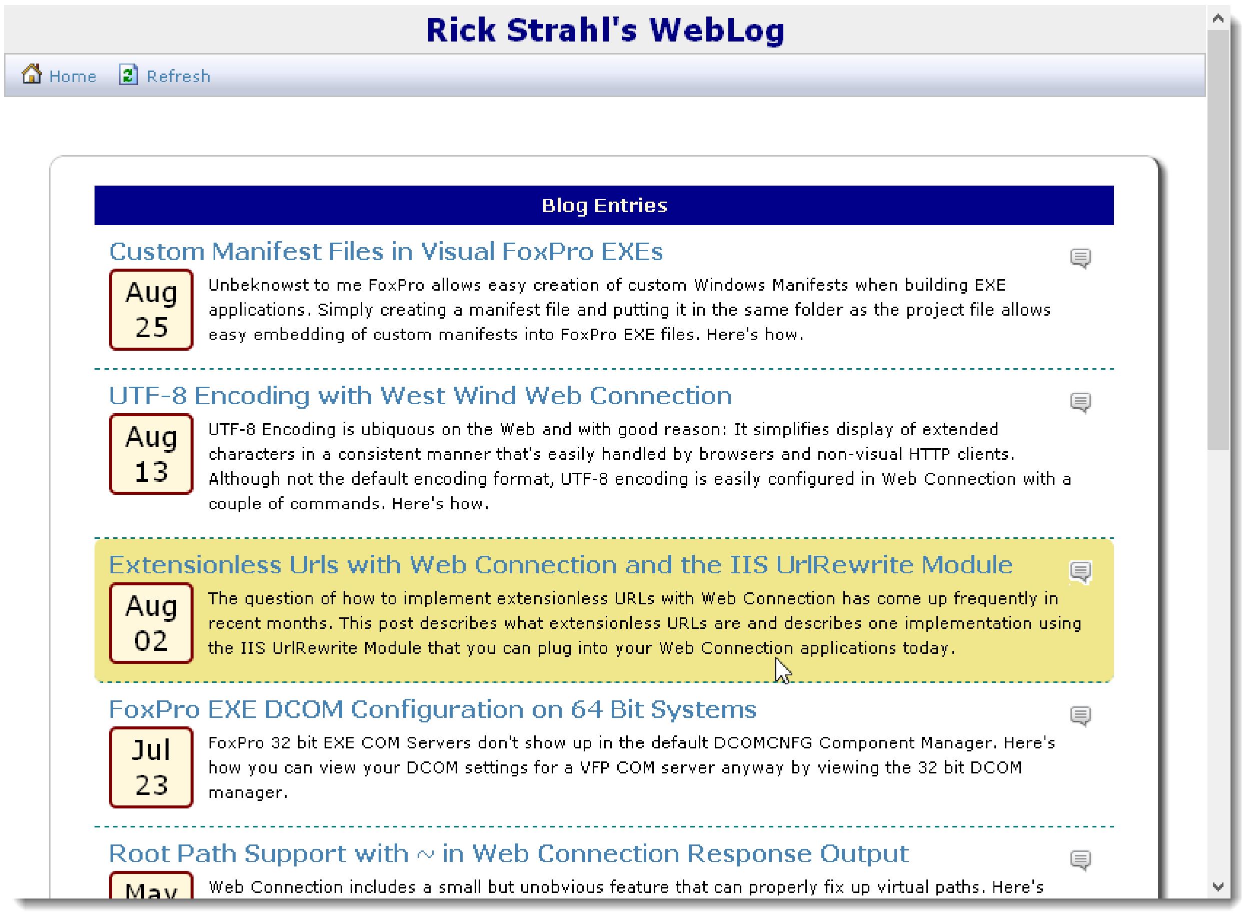
```
*************************************************************
DEFINE CLASS WebLogServer AS aspbase OLEPUBLIC
*************************************************************

************************************************************************
*  Init
****************************************
FUNCTION Init()

DODEFAULT()

SET PATH TO THIS.ApplicationPath + "weblog\;"

ENDFUNC
*   Init

ENDDEFINE
*EOC WebLogServer
```

There's no code in here beyond the initial configuration setup – we're actually going to use business objects loaded with ComHelper.CreateObject() for the most part so there will be very little or no code in this class. The main reason for this class is so that I have a top level object with the proper name (foxaspNet.WebLogServer) to identify this as my WebLog COM server.

The business objects live in WebLog_Business.prg and contains two classes called blog_Entry and blog_Comment. They inherit from wwBusiness and handle basic query and

CRUD operations, plus a few custom methods that retrieve data. I have to add this class to the project and then recompile.

So let's start with the WebLog List view similar to the one we built previously. Here's what the end result looks like:



Custom Manifest Files in Visual FoxPro EXEs
Aug 25
Unbeknowst to me FoxPro allows easy creation of custom Windows Manifests when building EXE applications. Simply creating a manifest file and putting it in the same folder as the project file allows easy embedding of custom manifests into FoxPro EXE files. Here's how.

UTF-8 Encoding with West Wind Web Connection
Aug 13
UTF-8 Encoding is ubiquous on the Web and with good reason: It simplifies display of extended characters in a consistent manner that's easily handled by browsers and non-visual HTTP clients. Although not the default encoding format, UTF-8 encoding is easily configured in Web Connection with a couple of commands. Here's how.

Extensionless Urls with Web Connection and the IIS UrlRewrite Module
Aug 02
The question of how to implement extensionless URLs with Web Connection has come up frequently in recent months. This post describes what extensionless URLs are and describes one implementation using the IIS UrlRewrite Module that you can plug into your Web Connection applications today.

FoxPro EXE DCOM Configuration on 64 Bit Systems
Jul 23
FoxPro 32 bit EXE COM Servers don't show up in the default DCOMCNFG Component Manager. Here's how you can view your DCOM settings for a VFP COM server anyway by viewing the 32 bit DCOM manager.

Root Path Support with ~ in Web Connection Response Output
May
Web Connection includes a small but unobvious feature that can properly fix up virtual paths. Here's

Figure 10 – Take 2 of our WebLog Entry display – now new and improved!

First let's add a new WebLogController.cs to the /Controllers folder.

C# - The Weblog entry Listing Controller method

```
public class WebLogController : Controller
{
    // property of controller class visible to all methods
    dynamic WebLog = ComHelper.CreateObject("foxaspnet.WebLogServer");

    public ActionResult Index()
    {
        // create a Blog_Entry Business object from FoxPro
        dynamic entry = WebLog.ComHelper.CreateObject("blog_entry");

        // Call business object method to retrieve cursor
        entry.GetEntries(10);

        // Convert the TEntries cursor to a collection
        dynamic entries = WebLog.ComHelper.CursorToCollection("TEntries");

        return View(entries);
    }
}
```

I've moved the creation of the FoxPro COM object into the constructor of the Controller because it's used in every method – might as well set it up once only so it fires every time the controller is created.

In the Index the first thing is to create the Blog_Entry business object. Note that I use ComHelper.CreateObject() which is very simple helper that simply does a RETURN CREATEOBJECT(lcProgId) to create a FoxPro object and return it over COM. Blog_Entry is not an OLEPUBLIC object in Fox, but any FoxPro object can be returned over COM and used by .NET.

This effectively passes a full business object with all the CRUD methods like Load(), Save() and an oData member for loaded data into the .NET code, so I can now call business object methods in the controller code. We'll look at the CRUD operations in the Show request a little later. Here I'm calling a custom GetEntries() method that creates a TEntries table internally. The method returns a count of records created which we don't care about here – only the cursor matters. As before the cursor is turned into a Collection using the ComHelper.CursorToCollection() method and then passed to the View for rendering.

Next let's create the View in /Views/WebLog/Index.cshtml:

C# Razor – Listing Web Log Entries with Embellishments

```
@{
    ViewBag.Title = "Index";
    Layout = "~/views/shared/_WeblogLayout.cshtml";
}

@section HtmlHeader{
    <style>
        a, a:hover, a:visited {
            text-decoration:none;
            color: Steelblue;
        }
        .blogentry {
            padding: 10px;
            border-bottom: dashed 1px teal;
            min-height: 70px;
        }
        .blogentry:hover {
            background-color: khaki;
            border-radius: 6px;
        }
        .blogtitle {
            font-weight: bold;
            font-size: 1.2em;
        }
        .blogabstract {
            font-size: 0.8em;
            margin: 5px 10px;
        }
        .contentcontainer{
            margin: 40px auto;
            width: 700px;
        }
        .datebox {
            float:left;
            width: 50px;
            border: solid 2px maroon;
            margin: 3px 10px 3px 0;
            padding: 2px;
            background: cornsilk;
            border-radius: 5px;
            font-size: 14pt;
            font-weight: normal;
            color: black;
            text-align: center;
            line-height: 1.3em;
        }
    </style>
}

<div class="toolbarcontainer">
    <a href="" class="hoverbutton"><img src="~/css/images/home.gif" /> Home</a>
    <a href="@Url.Action("Index")" class="hoverbutton"><img
src="~/css/images/refresh.gif" /> Refresh</a>
</div>


<div class="contentcontainer">

<div class="gridheader">Blog Entries</div>
@{ for (int i = 1; i <= Model.Count; i++)
   {
       dynamic entry = Model.item(i);

        <div class="blogentry" data-id="@entry.Pk">
            <div class="rightalign">
                <img src="~/images/comment.gif" class="hoverbutton" />
            </div>
            <div class="blogtitle">
                <a href="@Url.Action("Show", new { Id= entry.Pk
})">@entry.Title</a>
            </div>
            <div class="datebox">

                    @entry.Entered.ToString("MMM")<br />
                    @entry.Entered.ToString("dd")

            </div>
            <div class="blogabstract">
                @entry.Abstract
            </div>
        </div>
   }
}

</div>
```

This code is very similar to the last example. There are a few more embellishments in the display, but otherwise the logic is similar: The Razor code simply loops over the collection and displays each item. To try it out go to:

http://localhost/FoxProAspNet/weblog

or

http://localhost/FoxProAspNet/weblog/index

Displaying a WebLog Entry

Note that I created a link in the list that takes you to an individual blog entry with:

```
<a href="@Url.Action("Show", new { Id= entry.Pk })">@entry.Title</a>
```

I'm pointing at the Show method and passing in an Id parameter based on the Pk. The URL for this operation looks like this

http://localhost/FoxProAspNet/weblog/Show/890

This link still fails because we haven't implemented the Show method yet.

Now we need to load an individual blog entry. Let's create another Controller method called Show:

C# - Displaying an individual WebLog Entry using Fox Business Objects

```
public ActionResult Show(int id, CommentViewModel commentModel = null)
{
    dynamic entry = WebLog.ComHelper.CreateObject("blog_entry");

    // Load an Entry by it's Pk - entry.oData hold values
    if (!entry.Load(id))
        throw new ApplicationException("Invalid Id passed.");

    // on non-comment submission (GET) create a new ViewModel
    if (Request.HttpMethod != "POST")
    {
        ModelState.Clear();
        commentModel = new CommentViewModel(entry);
    }

    // Create a collection of comments
    dynamic comments = WebLog.ComHelper.CreateObject("blog_comment");
    comments.GetCommentsForEntry(id);

    // assign comments and actual entry data to model
    commentModel.Comments = WebLog.ComHelper.CursorToCollection("TComments");
    commentModel.Entry = entry.oData;
    commentModel.CommentCountText = entry.GetCommentCountText();
    // explicitly specify the Show View
    // so we can use it from other methods
    return View("Show",commentModel);
}
```

Here I create new blog_entry and blog_comment business objects. First I load up a blog entry by ID through the Load() method. Load() takes an ID and loads up the values of the WebLog data record into an *oData* member of the business object, so you have entry.oData.Title, entry.oData.Body and so on. Because the oData is an object that data can be passed to .NET easily.

A blog entry can also display related comments and so I use the blog_Comments business object to load up a cursor of comments, and then convert that cursor to a collection.

This page also allows a user to post a comment and I'm going to use MVC's ModelBinding to capture the input from a comment. ModelBinding allows properties of an object to be mapped to incoming POST data values from a form based on the name where the field name maps to a property on the model object.

Unfortunately model binding doesn't work with COM objects because .NET can't figure out the types used for objects dynamically as it can for native .NET types. So if I want to use model binding I can use a .NET class instead to capture the input and then assign the values as needed to the FoxPro COM object.

So in this Controller/View combination I'm doing something new: I using a custom ViewModel that holds the various FoxPro objects we need to access in the View as well as the input values from the Comment form.

ViewModels are a good practice in general anyway – they are meant to hold only the data the view needs and very little more. So rather than passing business objects to a view you are meant to pass ViewModels that hold only the data that is to be bound.

The .NET ViewModel I created for this form looks like this:

C# - A custom ViewModel that is passed to the Show View

```
public class CommentViewModel
{
    [Required]
    public string Title { get; set; }
    [Required]
    public string Author { get; set; }
    public string Email { get; set; }
    public string WebSite { get; set; }
    [Required, StringLength(2000, MinimumLength = 50)]
    public string Body { get; set; }
    public int Id { get; set; }
    public bool IsValid = true;
    public string Message;
    public string CommentCountText;

    // Entry oData instance
    public dynamic Entry = null;

    // Collection of Comments
    public dynamic Comments = null;

    public CommentViewModel()
    { }

    public CommentViewModel(dynamic entry)
    {
        Title = "re: " + entry.oData.Title;
        Id = (int)entry.oData.Pk;
        Email = string.Empty;
        WebSite = string.Empty;
        Body = string.Empty;
    }
}
```

The properties on top (with the {get; set; }) are the Comment input form values we'll use for model binding. The fields below – IsValid, Message, Entry and Comments are meant for data display only. This is the model that gets passed to the View.

The view that renders the output looks like this:

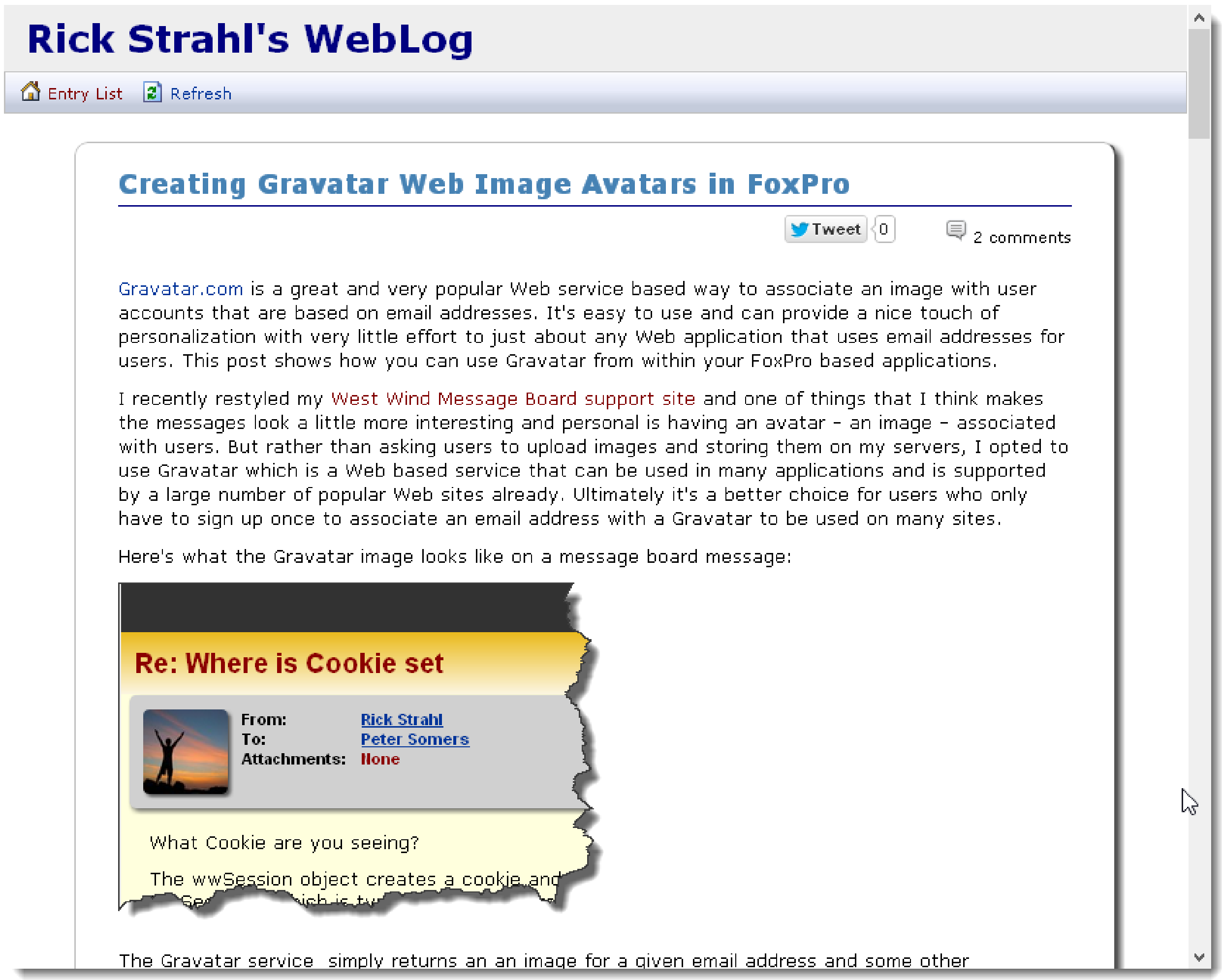
```
@model FoxProAspNet.Controllers.CommentViewModel
@{
    ViewBag.Title = Model.Entry.Title;
    Layout = "~/views/shared/_WeblogLayout.cshtml";
}
@section HtmlHeader
{
    <link href="~/Css/App/Show.css" rel="stylesheet" />
}

<div class="contentcontainer">
    <header>@Model.Entry.Title</header>
    <hr />
    <div class="subheader">
        <img src="~/images/comment.gif" /> @Model.CommentCountText
    </div>
    <article>
        @Html.Raw(Model.Entry.Body)
    </article>


    @if(Model.Comments.Count > 0) {
        <div class="gridheader">
            Comments
        </div>
        for(int i = 1; i <= Model.Comments.Count; i++)
        {
            dynamic comment = Model.Comments.item(i);

            <div class="comment">
                <div class="commentheader">
                    <b>@comment.Title</b><br />
                    <small><a href="@comment.Url">@comment.Author</a> -
                           @comment.Entered.ToString("MMM dd, yyyy")</small>
                </div>
                <div class="commentbody">
                    @Html.Raw(comment.body)
                </div>
            </div>
        }
    }

    @using( Html.BeginForm("SaveComment","Weblog")) {

            if (!string.IsNullOrEmpty(Model.Message))
            {
                <div class="errordisplay" style="width: 500px; margin: 20px
auto;">
                    <img src="~/css/images/warning.gif" /> &nbsp;
                    @Model.Message

                    @Html.ValidationSummary()
                </div>
                }
    <div id="CommentBox" class="contentcontainer" >


        <div id="CommentHeader">Leave your Mark</div>

        <table id="CommentTable">
            <tr>
                <td>Title:</td>
                <td>@Html.TextBox("Title", Model.Title)
                    @Html.ValidationMessage("Title")
                </td>
            </tr>
            <tr>
                <td>Your Name:</td>
                <td>
                    @Html.TextBox("Author", Model.Author)
                    @Html.ValidationMessage("Author")
                </td>

            </tr>
            <tr>
                <td>Your Email:</td>
                <td>@Html.TextBox("Email", Model.Email)</td>
            </tr>
            <tr>
                <td>Your Web Site:</td>
                <td>
                    @Html.TextBox("Website", Model.WebSite)
                </td>
            </tr>
        </table>

        <hr />
        <b>Comment:</b>
        @Html.TextArea("Body", Model.Body)
        <hr />
        <input type="submit" id="btnSaveComment" value="Save Comment"
class="submitbutton" style="width: 100px;"/>
    </div>
    <div>
        @Html.Hidden("Id", Model.Id)
    </div>
    }

    @if (!Model.IsValid || !string.IsNullOrEmpty(Model.Message))
    {
        // Scroll to bottom of page so messages can be seen
    <script type="text/javascript">
          $().ready( function() {
              $("#Title").focus();
          });
    </script>
    }

</div>
```

Notice the first line in the view:

```
@model FoxProAspNet.Controllers.CommentViewModel
```

This specifies that the view is strongly typed to the CommentViewModel passed in with Visual Studio providing Intellisense and compilation type checking in the View.

The top part of the view uses the Model.Entry object (which contains just the oData member of the business object) to display the actual Weblog Post information. A little lower the FoxPro Model.Comments collection is used to display the comments as a list. Here’s what the top of the form looks like:


Rick Strahl's WebLog

Entry List Refresh

Creating Gravatar Web Image Avatars in FoxPro

Tweet 0 2 comments

Gravatar.com is a great and very popular Web service based way to associate an image with user accounts that are based on email addresses. It's easy to use and can provide a nice touch of personalization with very little effort to just about any Web application that uses email addresses for users. This post shows how you can use Gravatar from within your FoxPro based applications.

I recently restyled my West Wind Message Board support site and one of things that I think makes the messages look a little more interesting and personal is having an avatar - an image - associated with users. But rather than asking users to upload images and storing them on my servers, I opted to use Gravatar which is a Web based service that can be used in many applications and is supported by a large number of popular Web sites already. Ultimately it's a better choice for users who only have to sign up once to associate an email address with a Gravatar to be used on many sites.

Here's what the Gravatar image looks like on a message board message:

Re: Where is Cookie set

From: Rick Strahl
To: Peter Somers
Attachments: None

What Cookie are you seeing?

The wwSession object creates a cookie and

The Gravatar service simply returns an an image for a given email address and some other

Figure 11 – The top of a Weblog Entry displayed in the browser

And finally on the bottom we have the comment form with its input values that are bound to the ViewModel properties.

Input fields are handled like this in MVC:

```
@Html.TextBox("Title", Model.Title)
@Html.ValidationMessage("Title")
```

This creates an input box that is automatically bound to the Model.Title property. When first displaying this textbox it shows the Model.Title value. After that it automatically picks up the user entered value. The ValidationMessage is optional, but it allows for messages that occur when validation of the model fails. If you look back at my model you'll see that several fields have a [Required] attribute, and the body field has [StringLength] attribute. If these validations fail, a validation message automatically is displayed below the input field.

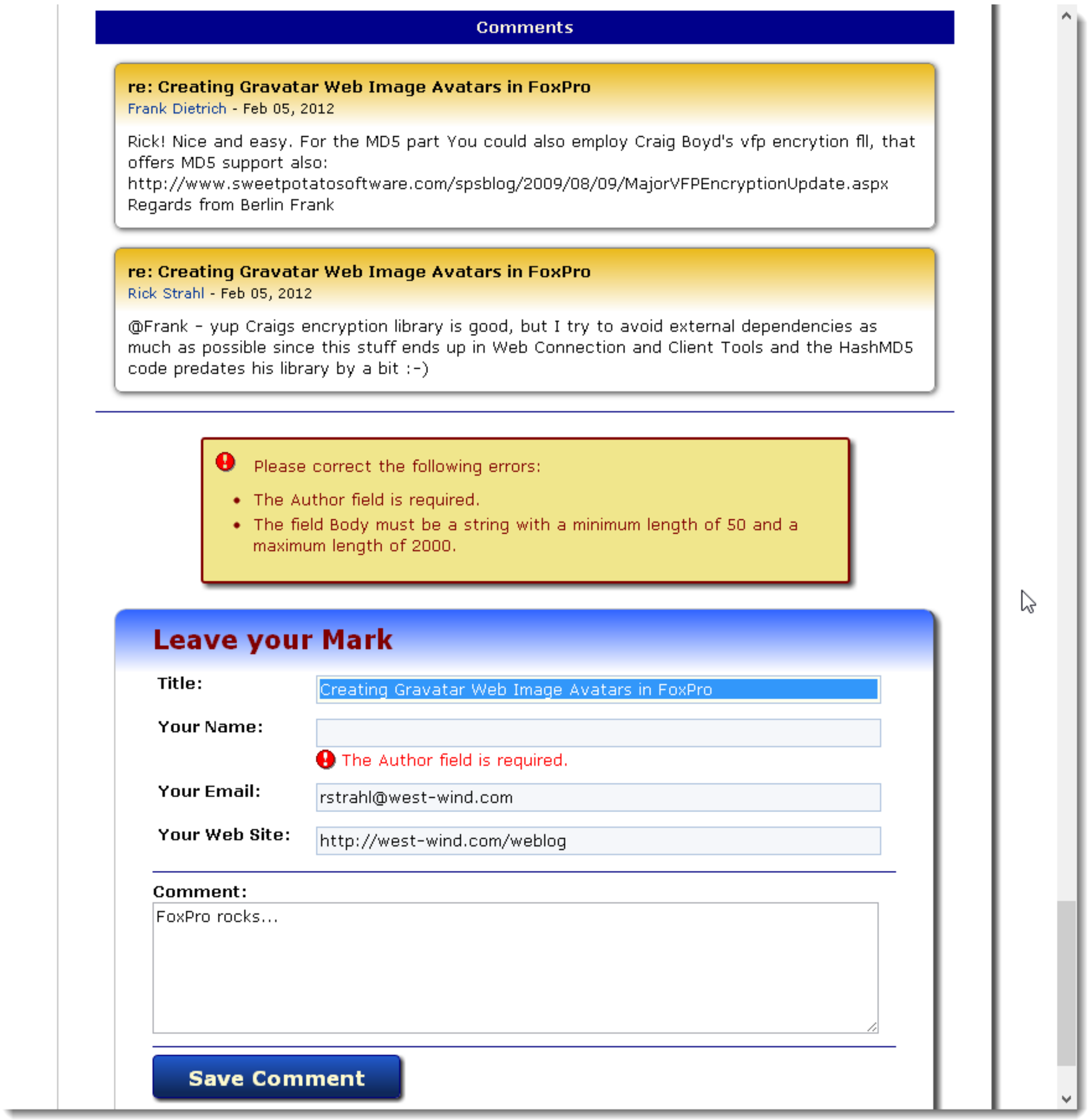
ASP.NET MVC tracks all POST data in its ModelState structure that also automatically binds parameters in controller methods that have matching parameters or object properties. On

POST operations the Model is validated and you can check ModelState.IsValid. When validation controls are on the form and the model is not valid the validation errors and optionally the ValidationSummary will display.

There's some logic that deals with displaying an error message in the View:

```
if (!string.IsNullOrEmpty(Model.Message))
{
      <div class="errordisplay" style="width: 500px; margin: 20px auto;">
            <img src="~/css/images/warning.gif" /> ;
                  @Model.Message
                  @Html.ValidationSummary()
     </div>
}
```

If I have an error message to display I conditionally display it in an error message box. Here's what the bottom part of the form looks like:



Comments

re: Creating Gravatar Web Image Avatars in FoxPro
Frank Dietrich - Feb 05, 2012

Rick! Nice and easy. For the MD5 part You could also employ Craig Boyd's vfp encrytion fll, that offers MD5 support also:
http://www.sweetpotatosoftware.com/spsblog/2009/08/09/MajorVFPEncryptionUpdate.aspx
Regards from Berlin Frank

re: Creating Gravatar Web Image Avatars in FoxPro
Rick Strahl - Feb 05, 2012

@Frank - yup Craigs encryption library is good, but I try to avoid external dependencies as much as possible since this stuff ends up in Web Connection and Client Tools and the HashMD5 code predates his library by a bit :-)

Please correct the following errors:

- The Author field is required.
- The field Body must be a string with a minimum length of 50 and a maximum length of 2000.

Leave your Mark

Title:	Creating Gravatar Web Image Avatars in FoxPro
Your Name:	
	The Author field is required.
Your Email:	rstrahl@west-wind.com
Your Web Site:	http://west-wind.com/weblog

Comment:
FoxPro rocks...

The final step left to handle is posting of a new comment. The process for this is to capture the input from the form via ModelBinding and then re-display any errors or a confirmation message to the user. Since we're redisplaying the original view there's no View associated with the save operation. We merely re-use the Show page.

To save a comment we use the following controller code:

```
[HttpPost]
public ActionResult SaveComment(CommentViewModel commentModel)
{
    if (!this.ModelState.IsValid)
    {
        commentModel.IsValid = false;
        commentModel.Message = "Please correct the following errors:";

        // call the show method explicitly
        return Show(commentModel.Id, commentModel);
    }

    dynamic comment = WebLog.ComHelper.CreateObject("blog_comment");

    // create new oData entity with defaults
    comment.New();

    // assign values from model to COM object
    comment.oData.Title = commentModel.Title;
    comment.oData.Author = commentModel.Author;
    comment.oData.Body = commentModel.Body;
    comment.oData.Email = commentModel.Email;
    comment.oData.Url = commentModel.WebSite;
    comment.oData.Entered = DateTime.Now;
    comment.oData.EntryPk = commentModel.Id;

    // save data to disk with Fox Com object
    if (!comment.Save())
        commentModel.Message = comment.ErrorMsg;
    else
    {
        // recreate an empty model to display
        int entryId = commentModel.Id;

        commentModel = new CommentViewModel();
        commentModel.Id = entryId;
        commentModel.Message = "Thank you for your thoughts!";

        // clear model state so data isn't displayed from it
        ModelState.Clear();
    }

    return Show(commentModel.Id,commentModel);
}
```

Notice the [HttpPost] attribute on the top which specifies that this method should only be called with a POST operation. The method accepts a CommentViewModel parameter as input. MVC tries to map the properties of this object to the names of fields from the page. So the Title field maps to the commentModel.Title property, and the hidden id field maps to the CommentModel.Id property.

When I submit the form this method is called because the View has the following Html form code:

```
@using( Html.BeginForm("SaveComment","Weblog")) {
```

Which creates a form tag that points at the SaveComments controller. In this method the first thing to do is check the model for validity. The model is valid as long as values can be matched and bound without failing validation of the model (the various validation attributes of the model) or fail type conversion. ModelBinding automatically attempts to convert values like numbers and dates to the matching property type. If a conversion fails a model error occurs.

ModelBinding only works against .NET objects – it does not work against COM objects, so if you want to use ModelBinding you have to create a .NET Model that holds the display data and then manually move the values between the model and your FoxPro objects as I do in the SaveComment code above.

ModelBinding is very cool, but without direct binding there's some extra work involved for FoxPro COM object – creating an explicit ViewModel and then pushing and pulling values to and from the model. Also keep in mind that MVC is still ASP.NET and the classic Request object still exists. If you don't want to use ModelBinding you can still use Request.Form["Body"] to retrieve a values or you can accept a FormCollection parameter:

```
public ActionResult SaveComment(FormCollection form)
{
    string body = form["body"];
```

In general though model binding is cleaner and meshes more closely with the MVC pattern where the model is not supposed to be a business entity but just a hold of display values.

When saving the controller code creates a new blog_comment business object, loads a new entity and then assigns the values from the ViewModel. When all are loaded the business object's Save() method which is part of the core wwBusiness methods called to write the data to disk.

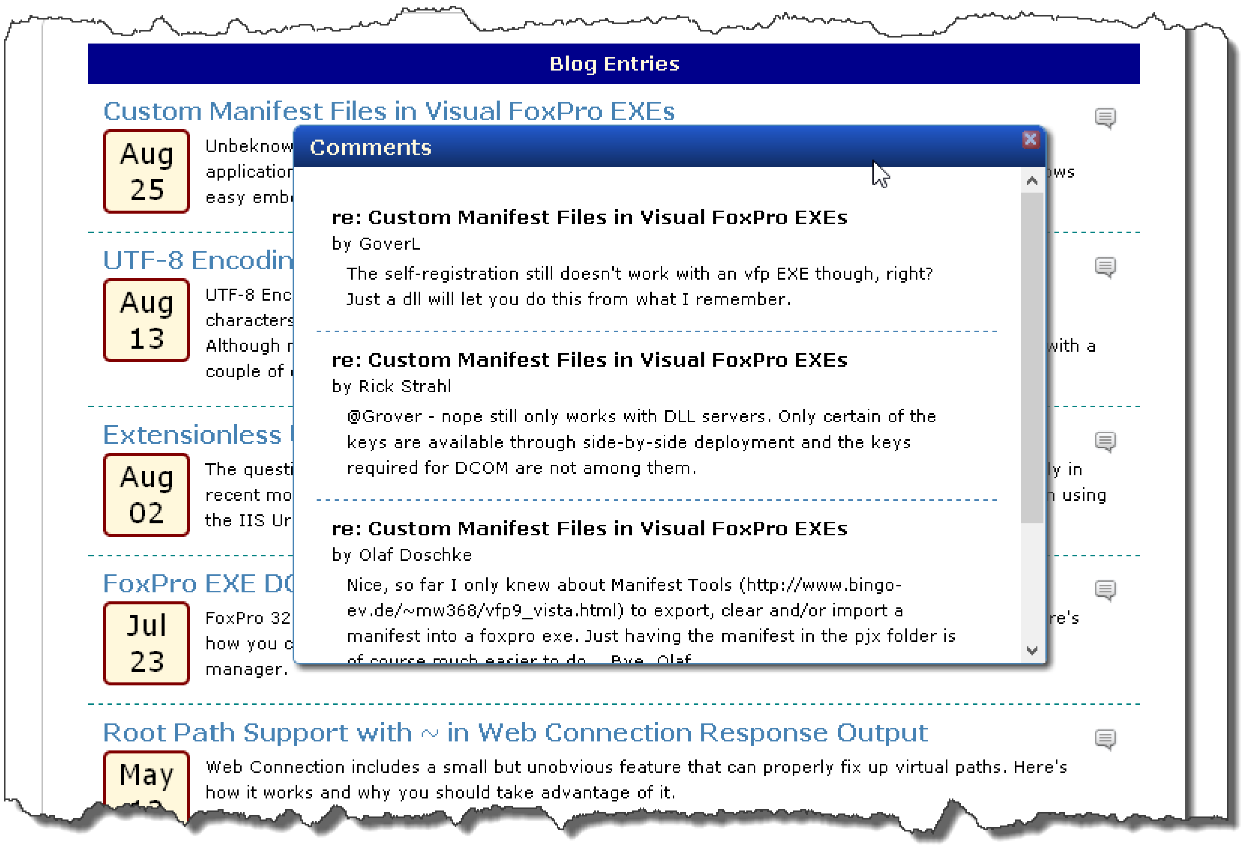
If there's an error the ViewModel gets an error message set which is then displayed when the page is redisplayed. Note that to redisplay the Show page I simply call the Show method and pass in the model and an id thereby reusing the existing display logic.

We've come full circle: We've listed some data, displayed an individual record and added some new data to the database all using FoxPro COM objects.

Adding some AJAX and JSON

One of the really nice things about MVC is that it makes it very easy to create a variety of different content. It's not just for HTML output but you can also very easily create JSON for AJAX applications.

Let's add a dynamic comment popup that is activated dynamically when the user clicks on the comment buttons on the entry list page. The result should look like this:


Blog Entries

Custom Manifest Files in Visual FoxPro EXEs

Aug 25

Unbeknow application easy emb

UTF-8 Encodin

Aug 13

UTF-8 Enc characters Although r couple of

Extensionless

Aug 02

The questi recent mo the IIS Ur

FoxPro EXE D

Jul 23

FoxPro 32 how you c manager.

Root Path Support with ~ in Web Connection Response Output

May

Web Connection includes a small but unobvious feature that can properly fix up virtual paths. Here's how it works and why you should take advantage of it.

Comments

re: Custom Manifest Files in Visual FoxPro EXEs
by GoverL

The self-registration still doesn't work with an vfp EXE though, right? Just a dll will let you do this from what I remember.

re: Custom Manifest Files in Visual FoxPro EXEs
by Rick Strahl

@Grover - nope still only works with DLL servers. Only certain of the keys are available through side-by-side deployment and the keys required for DCOM are not among them.

re: Custom Manifest Files in Visual FoxPro EXEs
by Olaf Doschke

Nice, so far I only knew about Manifest Tools (http://www.bingo-ev.de/~mw368/vfp9_vista.html) to export, clear and/or import a manifest into a foxpro exe. Just having the manifest in the pjx folder is

The logic here is pretty easy – we want to get the client to request all comments for a given blog post by id. Here's what the controller code for this JSON method looks like:

```
public JsonResult GetComments(int id)
{
    dynamic comment = WebLog.ComHelper.CreateObject("blog_comment");
    comment.GetCommentsForEntry(id);

    dynamic comments = WebLog.ComHelper.CursorToCollection("TComments");

    var commentList = new List<object>();
    for (int i = 1; i <= comments.Count; i++)
    {
        dynamic item = comments.Item(i);

        // create an anoymous type
        object cm = new {
            Title = item.Title,
            Body = item.Body,
            Author = item.Author,
            Email = item.Email,
            WebSite = item.Url,
            Id = item.Pk
        };

        commentList.Add(cm);
    }
    return this.Json( commentList, JsonRequestBehavior.AllowGet);
}
```

The code should be familiar by now: We're using the Fox COM object to create a blog_comment instance and call its GetCommentsForEntry() method which returns a cursor, that is then turned into a collection.

Notice that this method returns a *JsonResult* type, which can be easily created with the Controller's Json() method that takes an input of an object that is serialized into JSON.

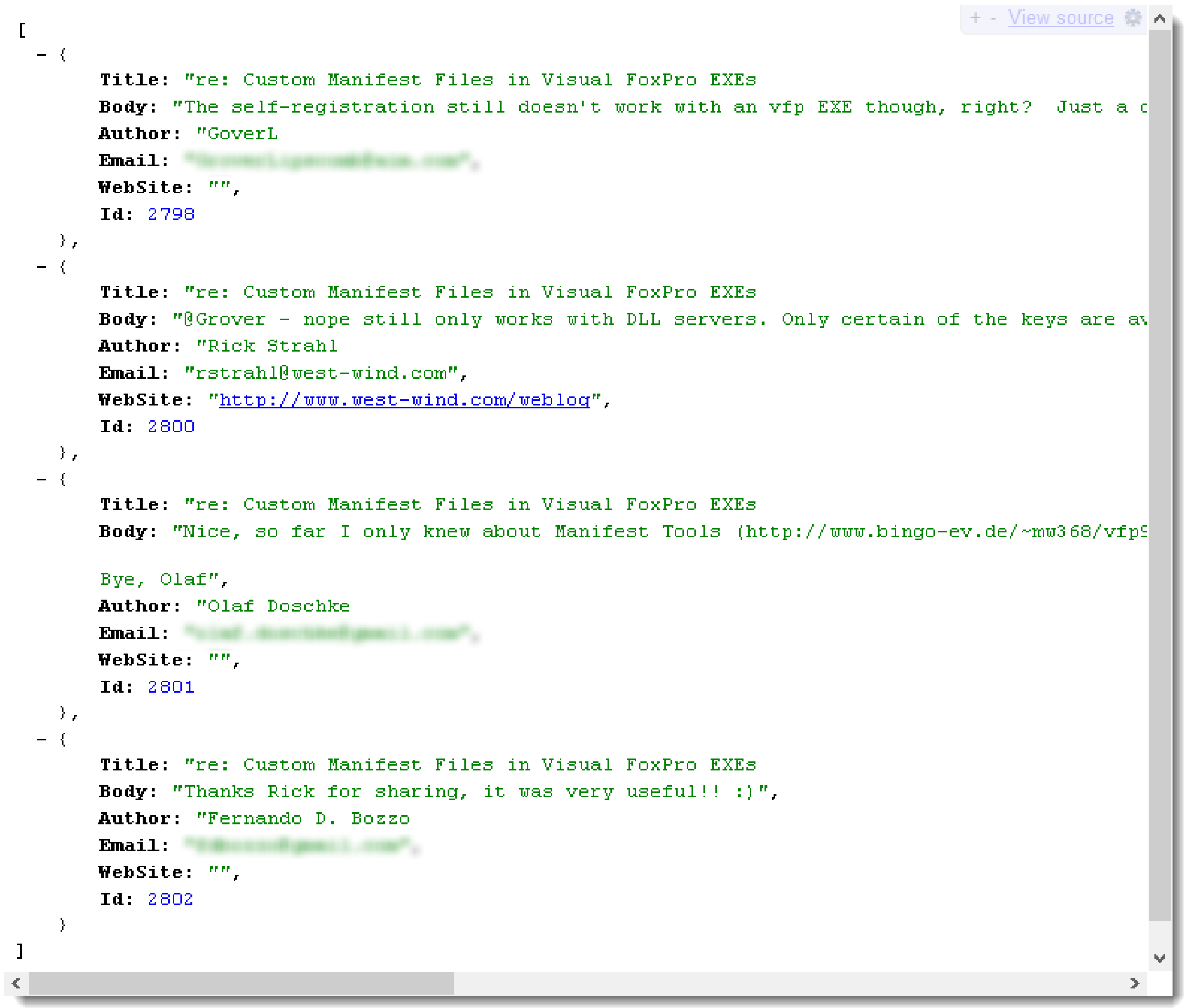
The good news is that MVC can easily convert objects to JSON. The bad news is that it only works with .NET objects, not with our FoxPro COM objects, so we can't just pass the COM comments instance to the Json() method. Instead we have to convert the FoxPro collection of objects into a .NET collection of objects first.

In this example I create an anonymous type called cm and assign just the properties I'm interested in. Each one of those objects is the added to a temporary List<object> instance that is serialized with the Json() method. You can use anonymous types or create explicit .NET types that hold the properties you are interested in – either way works.

You should now be able to hit the server a URL like this:

http://localhost/FoxProAspNet/Weblog/GetComments/890

and get a JSON response like this back:



+ - View source

```
[
  - {
        Title: "re: Custom Manifest Files in Visual FoxPro EXEs
        Body: "The self-registration still doesn't work with an vfp EXE though, right?  Just a c
        Author: "GoverL
        Email: [redacted]
        WebSite: "",
        Id: 2798
    },
  - {
        Title: "re: Custom Manifest Files in Visual FoxPro EXEs
        Body: "@Grover - nope still only works with DLL servers. Only certain of the keys are av
        Author: "Rick Strahl
        Email: "rstrahl@west-wind.com",
        WebSite: "http://www.west-wind.com/weblog",
        Id: 2800
    },
  - {
        Title: "re: Custom Manifest Files in Visual FoxPro EXEs
        Body: "Nice, so far I only knew about Manifest Tools (http://www.bingo-ev.de/~mw368/vfp9

        Bye, Olaf",
        Author: "Olaf Doschke
        Email: [redacted]
        WebSite: "",
        Id: 2801
    },
  - {
        Title: "re: Custom Manifest Files in Visual FoxPro EXEs
        Body: "Thanks Rick for sharing, it was very useful!! :)",
        Author: "Fernando D. Bozzo
        Email: [redacted]
        WebSite: "",
        Id: 2802
    }
]
```

Figure 12 – Comment listing output in JSON format

On the client side we can now use a bit of script code to retrieve the JSON result from the server when we click on one of the comment buttons.

I'll use jQuery and the Handlebars.js template parsing engine in this example to make the AJAX call and to render the JSON into HTML on the client.

Here's the script code:

Javascript – Retrieving and rendering Entry Comments with AJAX

```
<script src="~/Scripts/jquery.min.js"></script>
<script src="~/Scripts/handlebars.1.0.6.js"></script>
<script src="~/Scripts/ww.jquery.min.js"></script>

<script type="text/javascript">
    this.CommentTemplate = null;
    $().ready(function () {
          $("#CommentDisplay")
               .closable()
               .draggable({ handle: $("#CommentDisplay .dialog-header") });
          CommentTemplate = Handlebars.compile($("#CommentTemplate").html());

          $("img.hoverbutton").click(function () {
            // read id from .blogentry data-id attribute
              var id = $(this).parents(".blogentry").data("id");

            // now make JSON call
              $.getJSON("@Url.Action("GetComments")" + "/" + id,
                     function (comments) {
                         var html = CommentTemplate(comments);

                         $("#CommentDisplay .dialog-content").html(html);
                         $("#CommentDisplay")
                                .show()
                                .centerInClient();
                     });

              return false;
          });
      });
</script>
```

Let's start with the AJAX code which is the $.getJSON() function call. First I need to capture the ID of the element I clicked. I clicked on the image button, but I need to get the ID from the top level blogentry element Each blog entry renders with:

```
<div class="blogentry" data-id="@entry.Pk">
```

where the data-id contains the pk id of the post. I need to search backwards to find the .blogentry and retrieve its data-id value:

```
var id = $(this).parents(".blogentry").data("id");
```

Once I have the Id I can create the URL to make the JSON call. Here I use jQuery's getJSON() function which receives a URL and a result function that receives the resulting comment objects as an array. The handler code then uses Handlebars to render the template.

Rendering a Template with HandleBars.js=

Rendering the template with Handlebars is a two step process: You read the template which is embedded into the HTML document, and then render it with an object parameter that provides the data. Handlebars templates are HTML markup with expressions and a few structured statements embedded inside it. You can find out more about Handlebars' simple syntax from their Website.

The comment dialog is initially hidden on the pageand contains the handlebar template in the middle. The template can go anywhere, but if possible I like to embed it in the place where it goes when possible, so it's easy to place the context.

JavaScript Html – A HandleBars function embedded in an HTML snippet

```
<!--  Comment list HandleBars template -->
<div id="CommentDisplay" class="dialog boxshadow" style="width: 500px;display:
none;min-height: 240px;">
    <div class="dialog-header">Comments</div>
    <div class="dialog-content" style="max-height: 300px;overflow: auto;">
        <script type="text/handlebars" id="CommentTemplate">
            {{#each this}}
            <div class="comment" style="padding: 10px; border-bottom: dashed 1px
steelblue;" >
                <div><b>{{this.Title}}</b></div>
                <div class="small">by {{this.Author}}</div>
                <div class="small" style="padding: 3px 10px;">{{this.Body}}</div>
            </div>
            {{/each}}
        </script>
    </div>
</div>
```

First we have to retrieve the template from the document by using:

```
var html =$("#CommentTemplate").html();
```

and then compiling the template from the HTML. Handlebars takes the text of the template and turns it into a JavaScript function like this:

```
CommentTemplate = Handlebars.compile(html);
```

Templates should be compiled only once so typically the function is either a global variable (declared at window level) or inside of a global object/namespace. Here I use a global CommentTemplate that's declared at the top of the script tag.

Finally you can generate html from the compile template/function by calling the function with the data object as a parameter.

```
var html = CommentTemplate(comments);
```

This merges the data into the template. This particular template loops through each of the comment objects ({{#each this}}) and then embeds each of the values into HTML blocks ({{this.Title}} {{this.Author}} etc.). The end result of the function call is a raw HTML string that can now be embedded into the document.

Here I want to embed it into the CommentDisplay dialog into the contet area which can be accessed with:

```
$("#CommentDisplay .dialog-content").html(html);
$("#CommentDisplay")
    .show()
    .centerInClient();
```

The first line assigns the HTML using jQuery's html() function, while the second makes the floating window visible and centers it in the client. .centerInClient() is a jquery plug-in from my ww.jquery.js library which provides a host of very useful jquery plug-ins.

And voila, we now have a pop up JSON list. And with that our demos are done…

ASP.NET MVC and STA COM Components

One important thing I glossed over when I started these demos is that Visual FoxPro creates STA COM components. STAs are special components that have special threading requirements when running inside of a multi-threaded host like IIS. Threads that these components run on must be STA threads.

Unfortunately the only technology in ASP.NET that supports STA COM components natively is WebForms. MVC, Web Pages, ASMX Web Services, Web API and WCF all run only in **MTA** mode which is problematic for Visual FoxPro COM components.

If you create a controller method like this and run it:

```
public string ThreadingMode()
{
    return Thread.CurrentThread.GetApartmentState().ToString();
}
```

at **http://localhost/foxproaspnet/firstfox/ThreadingMode**

You'll get back **MTA** as the result value which is a problem.

Fortunately there's a workaround using a custom RouteHandler implementation. This RouteHandler implementation uses the WebForms Http Handler to provide the STA threading with three classes that can be thrown into a single class file and compiled.

C# - An STA compatible RouteHandler for ASP.NET

```
namespace FoxProAspNet {

public class MvcStaThreadHttpAsyncHandler : Page, IHttpAsyncHandler,
IRequiresSessionState
{
    RequestContext reqContext;

    public MvcStaThreadHttpAsyncHandler(RequestContext requestContext)
    {
        if (requestContext == null)
            throw new ArgumentNullException("requestContext");

        reqContext = requestContext;
    }
    public IAsyncResult BeginProcessRequest(HttpContext context, AsyncCallback cb,
object extraData)
    {
        return this.AspCompatBeginProcessRequest(context, cb, extraData);
    }

    protected override void OnInit(EventArgs e)
    {
        var controllerName = reqContext.RouteData.GetRequiredString("controller");
        if (string.IsNullOrEmpty(controllerName))
            throw new InvalidOperationException("Could not find controller to
execute");

        var controllerFactory = ControllerBuilder.Current.GetControllerFactory();

        IController controller = null;
        try
        {
            controller = controllerFactory.CreateController(reqContext,
controllerName);
            if (controller == null)
                throw new InvalidOperationException("Could not find controller: "
+ controllerName);

        }
        catch
        {
            throw new InvalidOperationException("Could not find controller: " +
controllerName +
                                                ". Could be caused by missing
default document in virtual.");
        }

        try
        {
            controller.Execute(reqContext);
        }
        finally
        {
            controllerFactory.ReleaseController(controller);
        }

        this.Context.ApplicationInstance.CompleteRequest();
    }

    public void EndProcessRequest(IAsyncResult result)
    {
        this.AspCompatEndProcessRequest(result);
    }

    public override void ProcessRequest(HttpContext httpContext)
    {
        throw new NotSupportedException();
    }
}


public class MvcStaThreadRouteHandler : IRouteHandler
{
    public IHttpHandler GetHttpHandler(RequestContext requestContext)
    {
        if (requestContext == null)
            throw new ArgumentNullException("requestContext");

        return new MvcStaThreadHttpAsyncHandler(requestContext);
    }
}

public static class RouteCollectionExtensions
{

    public static void MapMvcStaRoute(this RouteCollection routeTable,
                                      string name,
                                      string url,
                                      object defaults = null)
    {
        Route mvcRoute = new Route(url,
                                   new RouteValueDictionary(defaults),
                                   new MvcStaThreadRouteHandler());
        RouteTable.Routes.Add(mvcRoute);
    }
}
}
```

This code contains 3 classes: An HttpHandler implementation that provides the STA thread management that defers to the WebForms Page class for providing the actual STA processing, a custom route handler that basically invokes this HttpHandler and an optional RouteCollection extension that provides syntax that matches the standard MVC route handling logic.

The easiest way to enable the STA functionality is to use the extension method in the global configuration for ASP.NET in global.asax.cs or RouteConfig.cs.

Make sure the namespace is in scope or else the extension method won't be found:

```
using FoxProAspNet;
```

then to actually do the mapping hook up MapMvcStaRoute() in the RegisterRoutes() function:

C# - Registering the Mvc STA Route Handler via Extension Method

```
public static void RegisterRoutes(RouteCollection routes)
{
    // Web Resource Exclusion
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    // Custom Routing - uses STA threads
    routes.MapMvcStaRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index",
                        id = UrlParameter.Optional }
    );
    // Default MVC routing - uses MTA threads
    //routes.MapRoute(
    //    name: "Default",
    //    url: "{controller}/{action}/{id}",
    //    defaults: new { controller = "Home", action = "Index", id =
UrlParameter.Optional }
    //);
}
```

After you've hooked this up and the code compiles hit the test page again:

at http://localhost/foxproaspnet/firstfox/ThreadingMode

and you should get back **STA** this time. Happy times for your FoxPro COM objects.

You can find out more about ASP.NET and MTA threading in my Blog post here.

IIS and 64 Bit Operation

Most IIS servers and even Windows client machines these days are 64bit machines. If you are running any version of IIS7 it's very likely you are running a 64 bit version of IIS. If you do, make sure that you set up your application pool that hosts your FoxPro COM components to allow 32 bit operation.

To do this:

- Open the IIS Manager
- Go to Application Pools
- Find the application pool that your application runs in
- Right click and go to Advanced Settings
- Set the Enable 32-bit Applications to true
- Click OK and exit

If you don't do this you will see COM object creation errors failures as the 32 bit COM object can't be loaded into a 64 bit process.

Deploying ASP.NET and FoxPro Applications

One unfortunate issue with COM Interop development is that publishing COM components requires that the Web server (or at least the Web Application running your component) is shut down. When IIS runs and the COM component is active it is locked into memory and cannot be replaced, which means the update process for COM components requires administrative rights and often manual updating on the target server.

FoxPro COM servers also need to have the FoxPro runtimes installed in order to run. Unfortunately most ISPs do not have this in place. FoxPro COM objects also require full trust operation and again this is often not supported by public ISPs.

In general a COM Interop solution is mostly practical only in self-hosted environments or in full co-location scenarios where you control the server. Low cost ISP hosting with COM Interop is not likely to work with most ISPs due to the management and security overhead.

Summary

ASP.NET plus COM offers a stop-gap solution for integrating FoxPro and ASP.NET. Although the process can be a bit tedious with consistent design effort on creating self-contained business objects that can be easily called over COM, development with ASP.NET and FoxPro is a reasonable proposition especially since .NET 4.0 has come around.

With .NET 4.0 and dynamic types, accessing FoxPro COM components has gotten a lot easier especially with more complex objects or objects that are not defined in a type library. Many of the scenarios I showed here with ASP.NET MVC in the Web Log example would not have been easily done prior to .NET 4.0 and dynamics in .NET.

ASP.NET MVC provides a new mechanism for building ASP.NET Web applications and its controller and View based development mode is actually a good match for COM interaction – more so than WebForms was in the past. WebForms is problematic because most of WebForm's cool features like data binding and easy data updates don't work with COM objects. Likewise MVC's model binding also doesn't work with COM objects so some extra work is required to map FoxPro objects to .NET objects, but this is actually a good thing in MVC as that's the way the MVC pattern recommends things are done anyway.

FoxPro COM components always have to be aware of COM Threading issues in IIS and when not using WebForms some sort of custom STA thread generation scheme has to be used. For MVC a custom HttpHandler and RouteHandler combo provides a fairly self-contained mechanism to create STA threaded requests as shown in this article.

Perhaps the most vexing part of FoxPro COM Interop is debugging COM components if something goes wrong. Since there's no native FoxPro debugging support for FoxPro COM objects the best you can do is capture error messages and use application level logging to get additional information. It also doesn't help that dynamic object errors don't forward the FoxPro error message to .NET properly.

The key to working well in ASP.NET is to ensure that your FoxPro components are easily testable inside of Visual FoxPro, so if a problem arises in COM you can simulate a similar scenario with just a few lines of code. If COM operations to test a specific feature require more than 10 lines of code you probably don't have enough abstraction in your COM object models. You should always be able to test any COM operation or combination of operations fairly easily from the Command Window.

In the end FoxPro in ASP.NET is not a mission critical technology solution in my mind. While it works well and with the latest versions of .NET the process is even fairly smooth, the deployment and update issues, plus the limited debugging support make it less than optimal as a dedicated solution. Interop with ASP.NET should be treated as a stop-gap intermediate solution if you are migrating to .NET, or if you truly need to interoperate with other systems that are running .NET and need to access FoxPro data and business logic.

Resources

- **Examples for this article**
- **Calling VFP COM Components from ASP.NET** (old article)
- **Creating STA COM Compatible ASP.NET Applications**
- **Handlebars.js**