Creating and Consuming FoxPro Web Services
using .NET

Prepared for: Southwest Fox 2015
Updated: February 25th, 2016

By Rick Strahl
weblog.west-wind.com
rstrahl@west-wind.com

Materials:
https://bitbucket.org/RickStrahl/southwestfox2015_dotnetwebservices

Although today we have lots of choices on how to build services that connect applications over the Internet, **SOAP** (Simple Object Access Protocol) based services continue to be very popular with enterprise applications. I continue to see a ton of requests for either connecting FoxPro applications to existing SOAP based Web Services (most common) or for creating SOAP Web Services that let third parties connect and access data.

SOAP and WSDL

One of the main reasons SOAP is so popular for the enterprise is that it provides a fairly easy development model for developers. On the server side there are basic guidelines on how to create services. Typically you create classes and the service application that hosts the Web service can take these simple classes and turn them into a Web Service.

Likewise a client can connect to an auto-generated Web Service **WSDL** (Web Service Definition Language) document and create a client proxy that can then call the Web service. To the client application, that client typically just looks like an object with methods that return data from the service. You call a method and pass parameters, and you get a result back. The parameters can be simple, or as is often the case in Enterprise applications, can be very complex, hierarchical objects that describe huge enterprise application data models.

The nice thing about WSDL is that it can map these complex objects for you because the WSDL describes both the service method interface as well as all the related objects associated with each of the method parameters and result values. Think of WSDL as the source code for a sophisticated code generator that provides you with a class model.

SOAP and WS* Complexity

As nice as SOAP is, it has gotten very complex. In a way the simple origins of the SOAP protocol have been perverted and have been turned into an ugly mess with substandards. Nowhere is this more visible than the WS* extensions to SOAP which were supposed to be become SOAP 2.0, but which never materialized. WS* adds strict security, certificate signing, message validation and encryption support to messages and headers. WS* is wickedly complex and even with sophisticated tooling (like .NET's WCF framework) it can be very difficult to match up service host and service client behavior to use all the intricate configuration steps. Hope that the service you have to connect to is not a WS* service – otherwise plan on lots of extra time making the connection to the service work initially.

SOAP on Windows

SOAP 1.x is the original SOAP standard and it's still heavily used today. This is the simpler base standard, and if you build a Web Service you will most likely use this protocol. For FoxPro this protocol used to be supported via the SOAP Toolkit which is no longer supported by Microsoft and which was pretty crude to begin with. The SOAP Toolkit was Microsoft's last COM based interface to SOAP which was created in the late 1990s and discontinued since. There were also some SOAP extensions that supported some of the pre-cursors of the WS* protocols, but these are now hopelessly out of date.

Years ago I also built a wwSOAP client using FoxPro code. At the time it was built for SOAP 1.1 when things were still relatively simple, but with the increasing complexity of services built with SOAP 1.2, that tool also started to not work with many services. Parsing service definitions was very complex to account for all the strange XML conventions used especially by Java based services and I discontinued support for wwSOAP in 2009. While wwSOAP works, it often requires fine tuning of configuration to match specific namepsaces and additional headers. I would not recommend using this tool today.

All further SOAP based development from Microsoft was focused on the .NET framework. The original .NET framework shipped with an ASP.NET based SOAP service framework (**ASP.NET Web Services** or **ASMX)** as well as the WSDL.EXE tool that parses WSDL schemas into .NET classes. This tooling is still updated and it works well for a variety of use cases and it will be the focus of this paper and session.

In the mid 2000's Microsoft also introduces a new Web Service framework called **WCF** (Windows Communication Framework), which was a much more ambitious tool to handle all sorts of different kinds of services. Unlike the old ASP.NET Web Service and WSDL.EXE based clients, WCF can handle WS* Web services as well as a host of different protocols. WCF can run against HTTP, TCP, Named Pipes and even Memory Mapped files to do in-process processing of service requests. WCF is extremely powerful, but compared to the relative simplicity to ASP.NET Web Service WCF is practically Rocket Science.

As a FoxPro developer, you can use both of these technologies from FoxPro and I highly recommend that you do. While you might still be able to make the SOAP Toolkit or wwSOAP work, it'll probably require a bunch of tweaking to make it happen – especially if you deal with anything but very simple services.

You can create services either using ASP.NET Web Services or WCF and then either access FoxPro data directly using the OleDb database driver to FoxPro data, or by using FoxPro COM objects from the server applications. There are a few gotchas you have to watch out for – namely FoxPro's limited multi-threading capabilities in these environments (I'll talk about that later).

For the client, you can use the classic WSDL.EXE to generate .NET clients for Web Services or use WCF and import a Web Service, then call those generated proxy classes from FoxPro. This involves creating a .NET project and setting up a few wrapper method to create the service and then calling that from FoxPro. It help if you use the **wwDotnetBridge library** to connect to .NET and get around a number of limitations of basic COM interop between FoxPro and .NET.

As a general rule of thumb I suggest that if you're dealing with plain SOAP 1.x Web Services use the old .NET tooling of ASP.NET Web Services and the WSDL.EXE based proxies rather than WCF services or WCF Service proxies. The old Web service and client are much easier to use and implement and configure and manage. Use WCF whenever you need to deal with WS* protocol Web services.

In this article I'll describe how to create and consume SOAP 1.x services which is the simpler scenario but also the most common.

We'll look at:

- Creating Web Services with ASP.NET Web Services (ASMX)
- Returning FoxPro Data via OleDb
- Returning FoxPro Data via COM Interop
- Consuming Web Services with WSDL.exe Proxies
- Using wwDotnetBridge to call .NET Web Service Proxies

Creating Web Services with ASP.NET Web Services (ASMX)

ASP.NET Web Services, also known as ASMX due to the extension that's used, was introduced with the first version of .NET. It's an ASP.NET based implementation of Web Services that's purely grounded in the HTTP protocol. It's implemented as a custom ASP.NET HttpHandler that's been a solid SOAP 1.x platform since the beginning of .NET.

The beauty of this platform is that it's very easy to use and get started with. There's no configuration of any kind required, you simply create a new ASMX file, create a class that has a few special attributes to mark the service and service methods and you have a Web service.

Let's create a new .NET Solution in Visual Studio.

- Open Visual Studio 2015 (or 2013)
- New Project
- Create new ASP.NET Web Application
- Name it AlbumViewerServices
- Create Empty Project

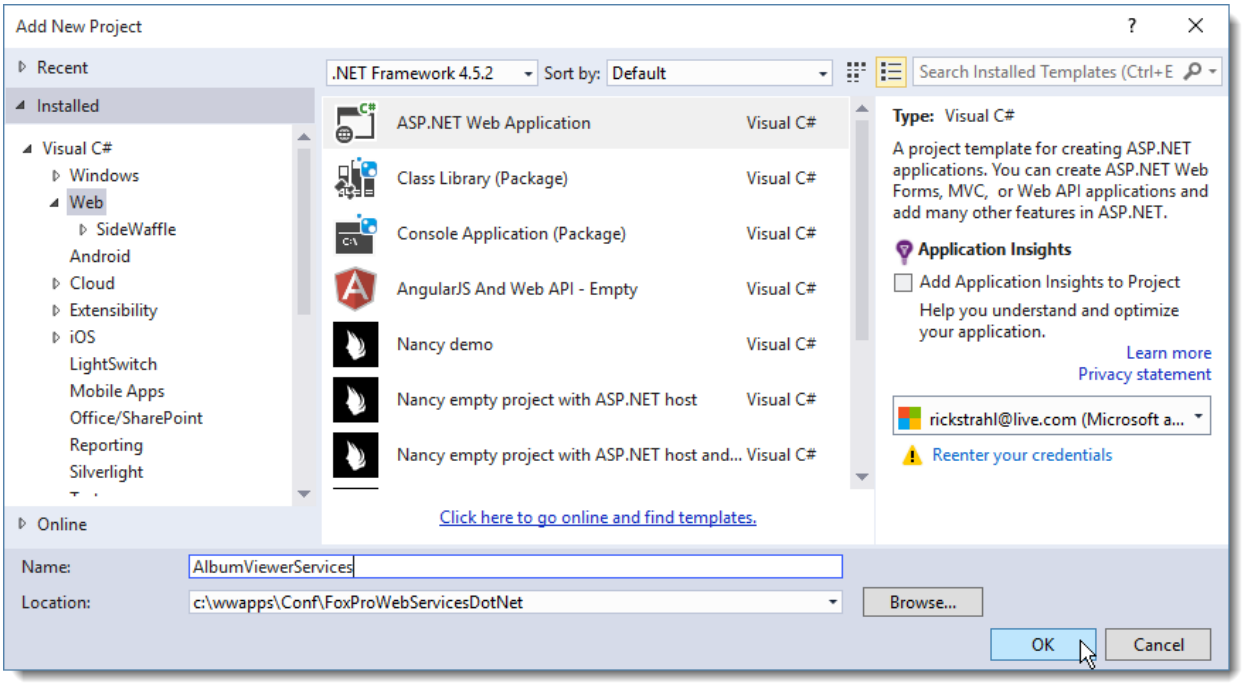


Figure 1 – Creating a new Web Application Project in Visual Studio

This creates a new project and now let's add an ASP.NET Web Service:

- On the new project right click Add | New Item
- Type ASMX into the searchbox
- Select Web Service (ASMX) from the list
- Click Add

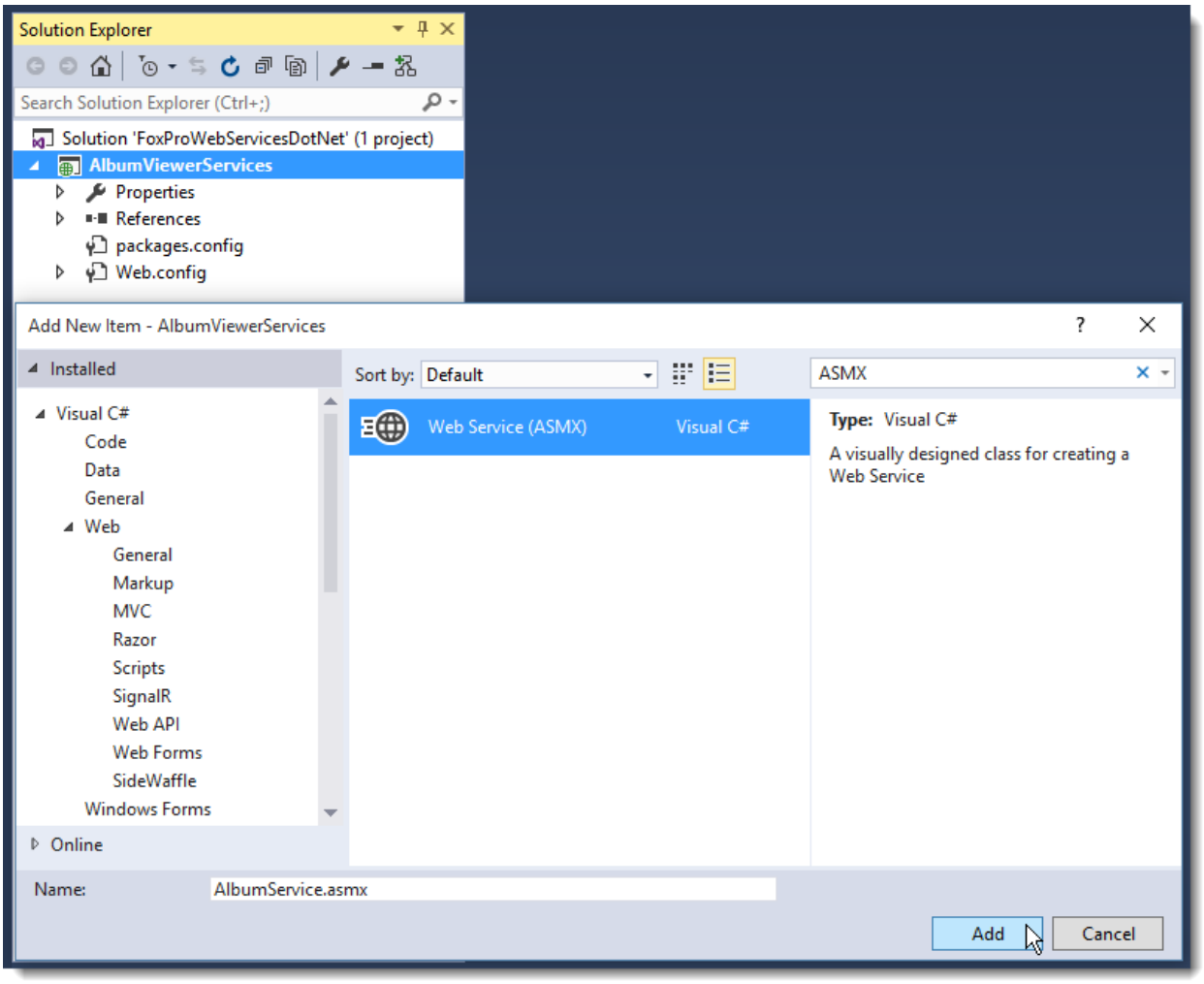


Figure 2 - Adding an ASMX Web Service to the Web project

This adds a new Web Service that includes the ASMX 'markup' file that is distributed and a 'code-behind' file that is actually a class that implements the Web Service.

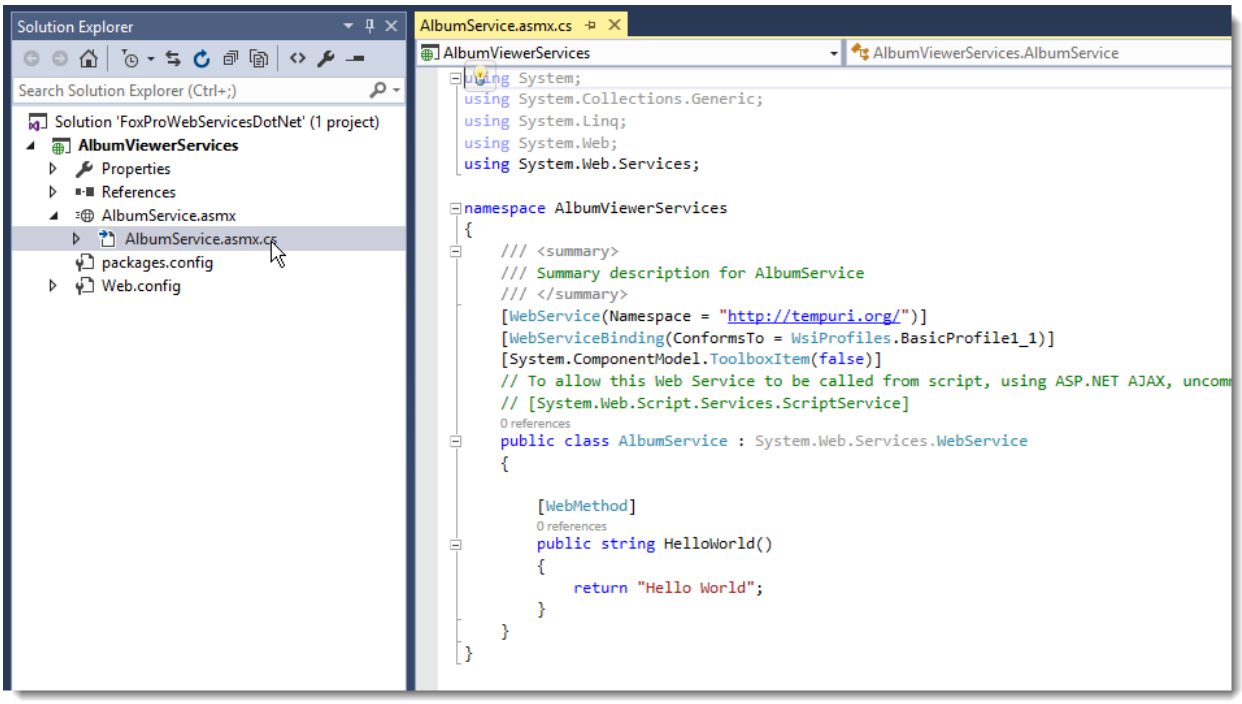


Figure 3 – The generated ASMX Web Service is a class with Attributes to identify Web methods to expose to the service.

The ASMX file is merely a shell that points that codebehind and provides the Web server a URL to access the service with.

At this point you actually have a working Web service without doing anything else at all. To launch you can either start the Web Server by running your project (F5 or the Run button) or by right clicking on the Web Service and use ***View in Browser***.

This brings up your selected browser on the Service description page:

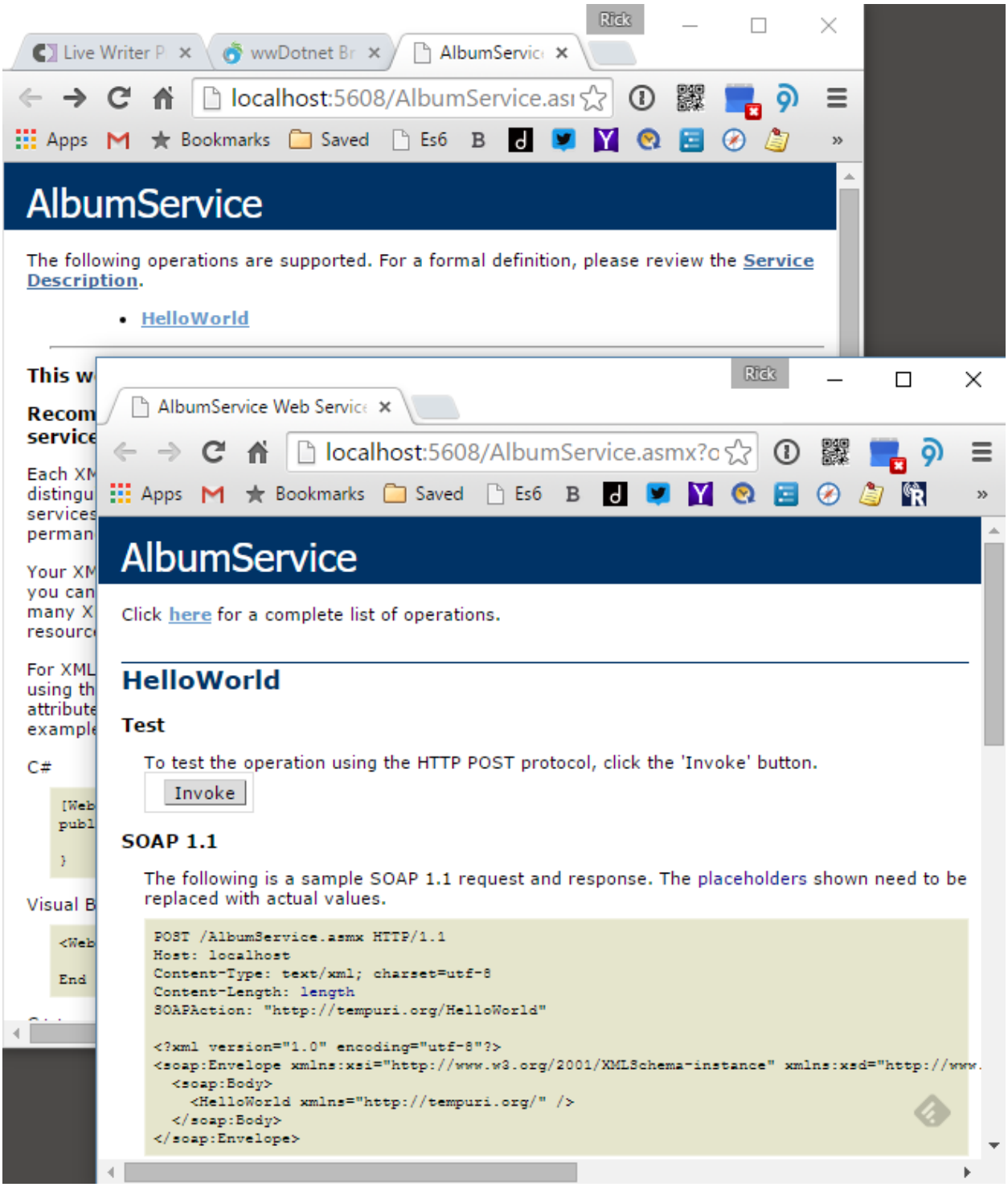


Figure 4 – A Web Service contains a test page you can use to test simple methods by invoking them interactively.

The sample page lists out all the available methods that are available on the service as well as providing a link to the WSDL document – which is basically the same URL with ?WSDL appended: **http://localhost/AlbumViewerServices/AlbumService.asmx?WSDL**

You can now click on any of the methods and then quickly test the methods that are provided by the service. If there are simple parameters the test page allows you to plug in values for the parameters. This only works for simple types like strings, numbers, Booleans etc. The page also shows examples of the XML trace for a request that breaks out the structure of the parameters passed. So complex objects are simulated with deserialized XML traces.

The nice thing about this is that you have to do nothing to get this functionality. Both WSDL document and the sample page are updated everytime you change the service – it's fully automatic.

The Generated Service

```
[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
[System.ComponentModel.ToolboxItem(false)]
public class AlbumService : System.Web.Services.WebService
{

    [WebMethod]
    public string HelloWorld()
    {
        return "Hello World";
    }
}
```

A WebService is a class that has a **[WebService]** Attribute and methods that are marked up with the **[WebMethod]** attribute to identify the actual service endpoints. Each method that has the latter attribute will be exposed in the WebService – all other methods public or private alike will not.

All you have to do to add additional methods to this service is simply add another method with **[WebMethod]** attribute and you're set.

There are some limitations on what you can return however. Objects and collections have to be serializable by the XmlSerializer in .NET and you cannot return Interfaces or objects that contain members that are interfaces – all members of objects have to be concrete types and have to have parameterless constructor so they can be instantiated generically by the serializer.

But for creating services that proxy FoxPro data that is unlikely to be an issue as you have to explicit create the classes to expose anyway.

Fix the Service URI

One thing you'll want to change is service URL which defaults to *tempuri.org*. While it's a valid URL it's not unique – you can bet lots of people publishing services forget (or don't know) to change the namespace of the Web service to something that represents the service uniquely.

A Uri is unique identifier – it doesn't have to match a real URL on the Web but it should be unique and identify the application. Usually the application name is a good one to use. So I'll use:

```
[WebService(Namespace = "http://west-wind.com/albumservice/asmx/")]
```

The typical usage is to use the company domain (if you have one) and then something that identifies the service. But it doesn't matter as long as the Uri use valid URL syntax – ie. use a moniker (http://) and something that can be interpreted as a domain name or IP address.

Switch to IIS

By default the project is create using IIS Express. This works, but in order to test the service you have to make sure that IIS Express is actually running. You also get a funky service URL with a port number that's not accessible across the network. For service applications you'll want your service to always be running

and for this reason I prefer running inside of IIS rather than IIS Express. That way I don't have to worry about whether IIS express is running or not. Note that in order to use full IIS for debugging you have to run Visual Studio as an Administrator.

You can create a Virtual Directory in side of Visual Studio by simply bringing up the Project Web settings and using the Create Virtual Directory.

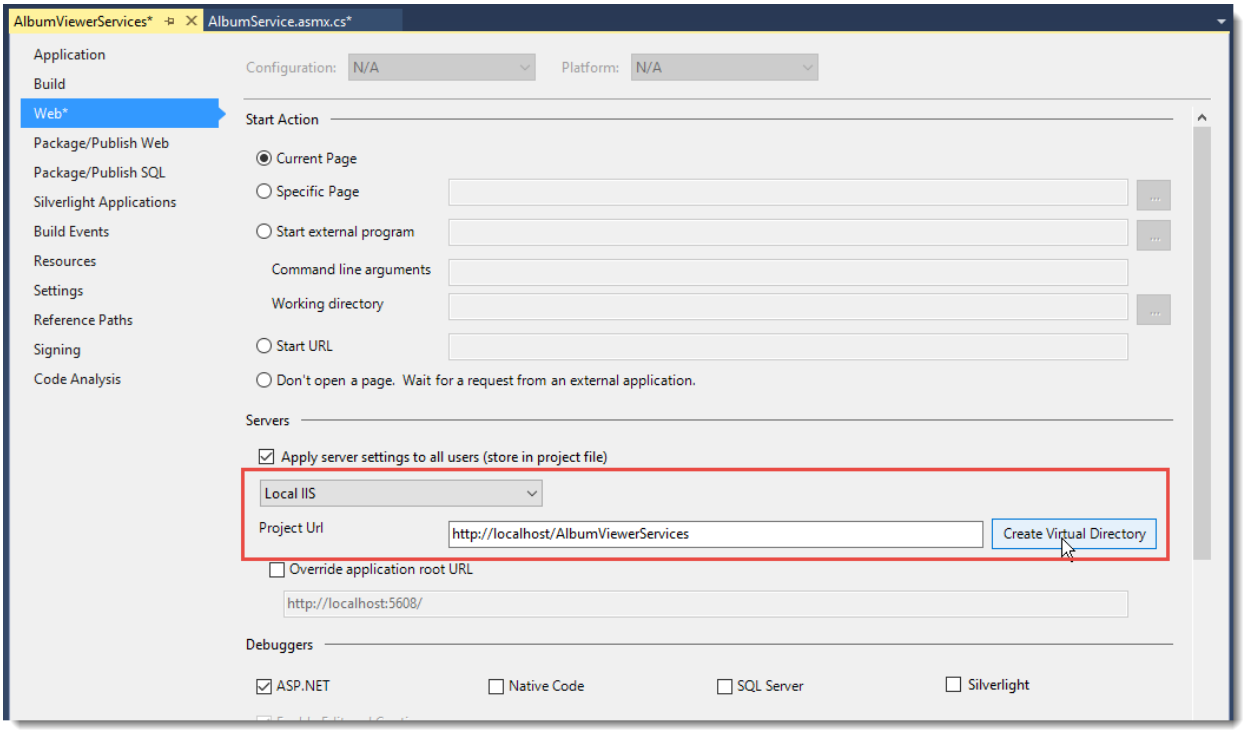


Figure 5 – Telling Visual Studio to use IIS instead of IIS Express makes debugging easier. Just remember that you need to run as an Admin in order to debug using full IIS.

The service Url now changes from:

http://localhost:5608/AlbumService.asmx

to

http://localhost/AlbumViewerServices/AlbumService.asmx

but it will always be up and running (assuming IIS is installed). Note that you can use IIS Express if you prefer, just remember you have to start it in order to get the service to run.

Test the Service

The easiest way to test the service is to just use the Service page to bring up the service and test one of the methods.

To do this we need to compile the code in Visual Studio (Ctrl-Shift-B) and then we can use View In Browser or start to debug to bring up the service page.
So let's make a slight change to the service so we can see how to modify code and see the change.

```
[WebMethod]
public string HelloWorld(string name = null)
```

```
{
    return "Hello " + name + ". Time is: " +
           DateTime.Now.ToString("MMM dd, yyyy  hh:mm:ss");
}
```

If we now compile and open this up in the browser we'll see:

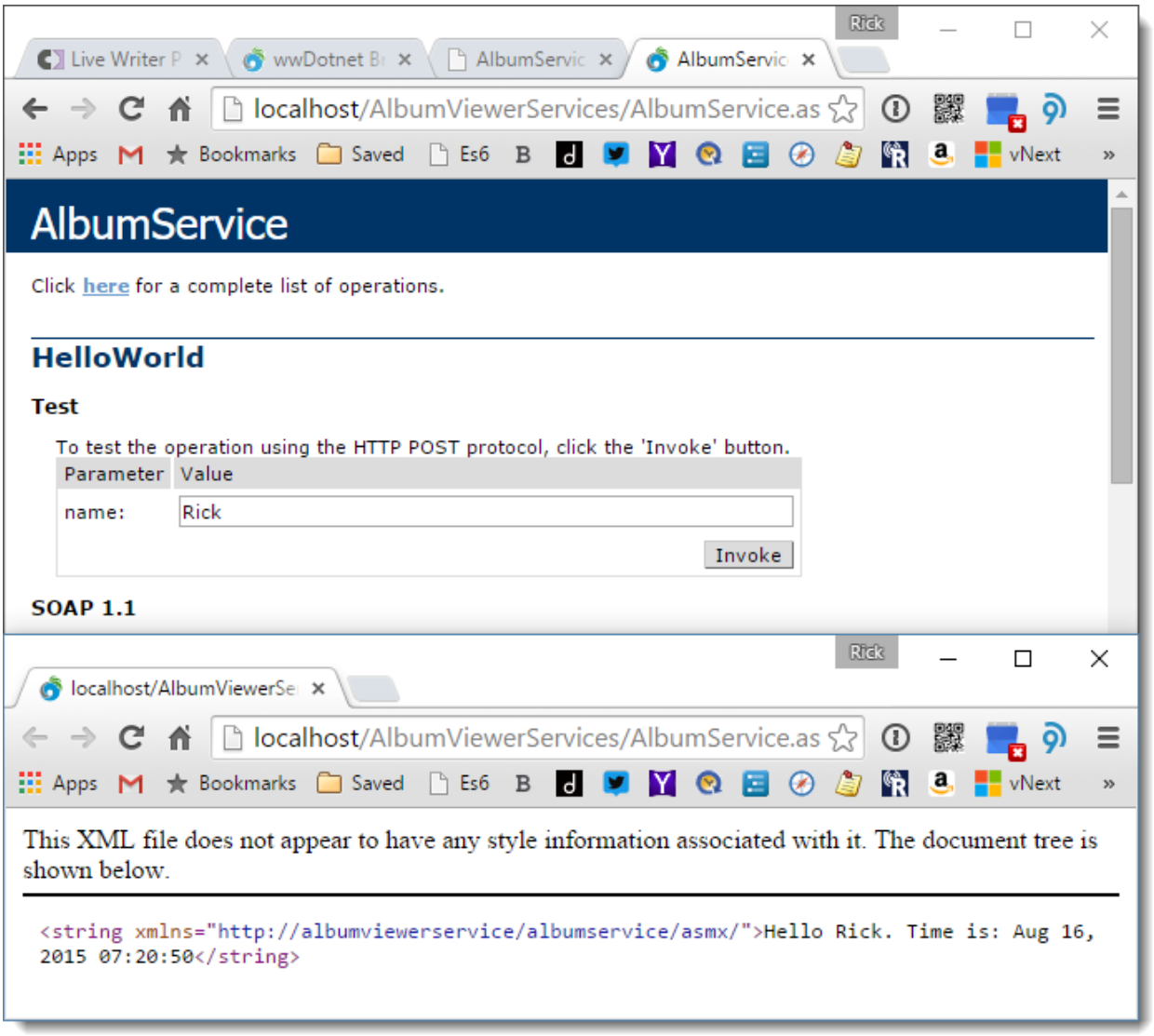


Figure 6 – Testing your first .NET Web Service.

Voila – our first customized Web Service method

In this example, I simply use a method with a simple input parameter and result value – in this case both strings. These are easy to create and it's super easy to create methods like these.

However, most Web Services are bound to have complex objects associated with with it and a little more effort is required to create services that return complex types.

Creating our AlbumService

So we're going to create an AlbumService, that can retrieve Artists, Albums and Tracks. In order to create a proper .NET service we'll need to write some basic .NET code to create the classes that the service requires for input parameters and output result values.

Specifically we'll need to create the Artist, Album and Track classes so that the service can return these objects as proper XML structures.

Don't return XML Strings!

A common mistake by developers new to Web Services is to think that you can avoid having to create a nice interface by simply returning an XML string and get the structure you need that way. While you can certainly push XML strings through a Web service – they are strings after all – it's extremely bad form to do so because as soon as you return a XML in a string the schema information that goes into the WSDL definition for the Web service is lost. This means anybody consuming the Web service will only know that you are returning a string, but not what's actually in the string.

So the proper way to return complex results is to return proper objects and collections from the actual service methods. In order to do this, we'll need to create these objects that we plan on returning.

> ***Note:***
> *Later we'll look at returning data FoxPro COM objects, but .NET cannot serialize objects that originate in FoxPro. Only .NET objects can be serialized – everything else has to be parsed into .NET objects for serialization.*

So let's see what this process looks like.

Creating a Model

The next step is to see how to create a new method. So this service I'm going to create will retrieve some album and artist data from a database. We have Artists, Albums and Tracks and that data will be returned.

We'll start by creating the messaging classes which match the data structure of the FoxPro tables I'm accessing. Here are the three classes that define the structure:

```
public class Artist
{
    public int Id { get; set; }

    public string ArtistName { get; set; }
    public string Description { get; set; }
    public string ImageUrl { get; set; }
    public string AmazonUrl { get; set; }
}

public class Album
{
    public int Id { get; set; }
    public int? ArtistId { get; set; }

    public string Title { get; set; }
    public string Description { get; set; }
    public int Year { get; set; }
    public string ImageUrl { get; set; }
    public string AmazonUrl { get; set; }
    public string SpotifyUrl { get; set; }
    public virtual Artist Artist { get; set; }
    public virtual Track[] Tracks { get; set; }
}

public class Track
{
    public int Id { get; set; }
```

```
    public int? AlbumId { get; set; }

    public string SongName { get; set; }
    public string Length { get; set; }
    public int Bytes { get; set; }
    public decimal UnitPrice { get; set; }
}
```

These are the classes we'll use as input and output messages for our service methods.

Next let's create a method that returns a list of artists – for now just created by hand in .NET code so we can see the logic of creating and populating our model objects and returning the data:

```
[WebMethod]
public Artist[] GetArtists()
{
    var artists = new List<Artist>();

    artists.Add(new Artist()
    {
        Id = 0,
        ArtistName = "Foo Fighters",
        Description = "Foo Fighters are an American Band",
        ImageUrl = "http://media.tumblr.com/tumblr_mb76f02FkJ1qfo293.jpg"
    });

    artists.Add(new Artist()
    {
        Id = 1,
        ArtistName = "Motörhead",
        Description = "Motörhead have never JUST been the best rock'n'roll band in the world.",
        ImageUrl = "http://alealerocknroll.com/wp-content/uploads/2014/07/motorhead.jpg"
    });

    return artists.ToArray();
}
```

If we now compile and go over to the test page we get a result like this from the Test page:

```
<ArrayOfArtist xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://albumviewerservice/albumservice/asmx/">
  <Artist>
    <Id>0</Id>
    <ArtistName>Foo Fighters</ArtistName>
    <Description>Foo Fighters are an American Band</Description>
    <ImageUrl>
      http://media.tumblr.com/tumblr_mb76f02FkJ1qfo293.jpg
    </ImageUrl>
  </Artist>
  <Artist>
    <Id>1</Id>
    <ArtistName>Motörhead</ArtistName>
    <Description>
      Motörhead have never JUST been the best rock'n'roll band in the world.
    </Description>
    <ImageUrl>
      http://alealerocknroll.com/wp-content/uploads/2014/07/motorhead.jpg
    </ImageUrl>
  </Artist>
  …
</ArrayOfArtist>
```

Cool, we got a working Web service that returns a static list of artists. Easy!

Well – sort of. The HTML test page actually isn't making a real Web Service call. Rather it's testing the service by using a POST request pushing values to the methods we've created in the service and then serializing the result. This is very useful for quick testing, but we're really not using SOAP in order to make this happen.

SOAP UI

A better way to test a service and to actually get a feel for what messages to the service look like you might want to test the service with a tool like SoapUI. Soap UI is a Web Service test tool that you can point at a WSDL document and it will figure out the service structure and give a testable XML snippet you can use to send SOAP messages to the service. IOW, it lets you properly test the service as SOAP.

To use SoapUI start it up and create a new Project. When prompted provide the WSDL URL to the service which is:

http://localhost/AlbumViewerServices/AlbumService.asmx?WSDL

Here's what the SOAP UI traces look like for the HelloWorld and GetArtists service methods:

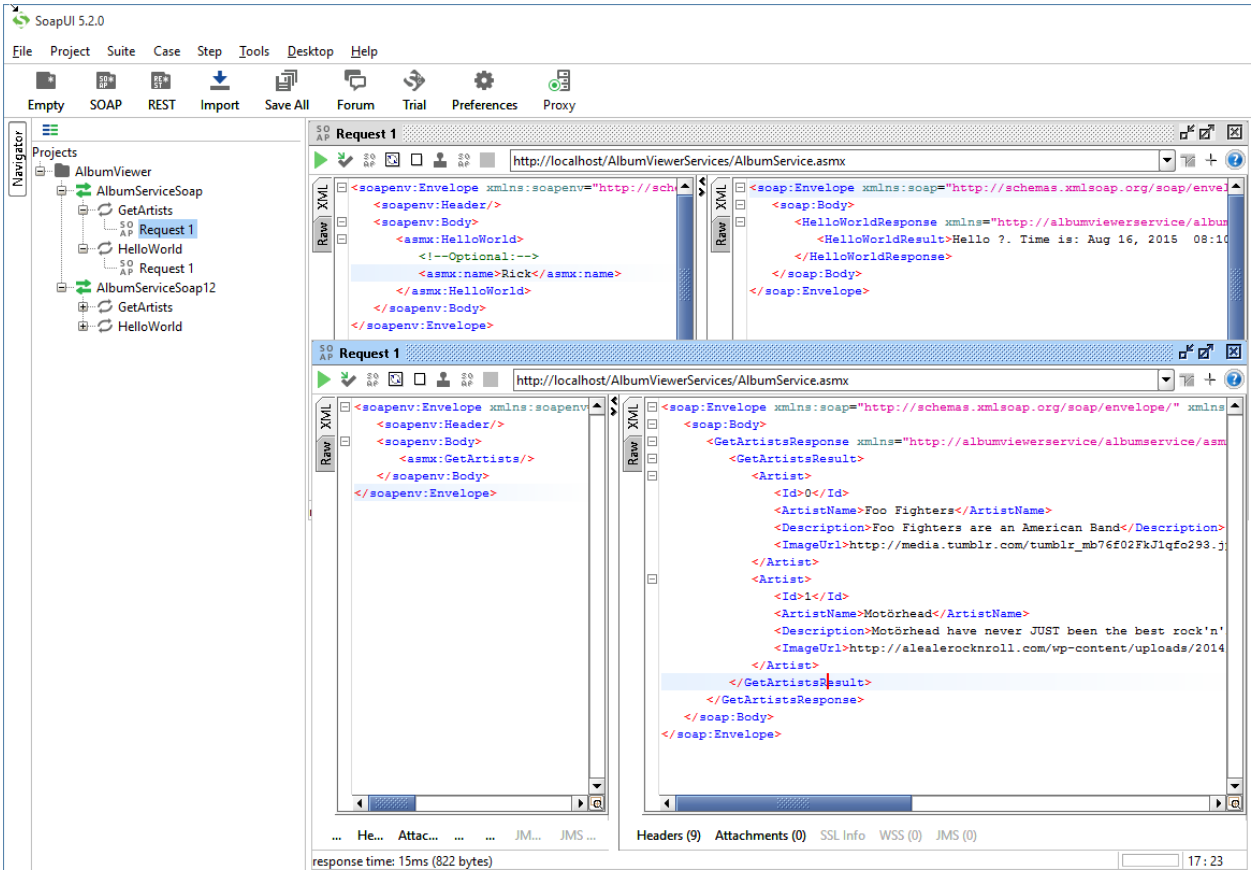


Figure 7 – Using SoapUI to test SOAP requests lets you examine full SOAP traces and save and replay those requests with data you enter.

Soap UI looks at the WSDL and creates empty request traces for you where the values for the inbound parameters (including nested objects when you have them as inputs) are stubbed out with ? characters. You can fill in the ? to test the service with live values.

Here you can see the SOAP requests made both by the client to request the data and by the server returning the data. One of the reasons SOAP UI is nice is that you can save a project and so have a simple and easily repeatable way to test various SOAP methods and see the results. At this point we don't have a Service client yet, so this functionality is useful while we build our servers and later our clients.

FoxPro Data?

Ok, so now we have the basics of the service in place – we can create Web Service methods and we can return data. We have our data structures that we need to fill, but now we actually need to get the data. Since we're talking about FoxPro here, there are a few different ways to get at the data:

- Access data directly using the .NET OleDb Driver
- Access data via COM object with .NET Interop

FoxPro OleDb Driver

The FoxPro OleDb driver lets you access FoxPro data from disk directly. The driver is not very high performance as it has threading issues similar to the FoxPro runtimes, but it works fairly well for low to medium level performance levels and especially for reading data.

Using the Fox OleDb provider means you'll have to write some data access/business logic in .NET code and format the data returned from queries into the .NET objects.

I'm going to use some helper classes in the free Westwind.Utilities .NET library which provides a simple data access layer.

First let's refactor the code a little so we can switch back and forth between different modes of retrieving the data.

```
[WebMethod]
public Artist[] GetArtists()
{
    if (App.Configuration.ProcessingMode == ProcessingModes.Dotnet)
        return GetArtistsStatic();
    else if (App.Configuration.ProcessingMode == ProcessingModes.FoxProOleDb)
        return GetArtistData();

    return null;
}
```

Next add a method that gets the data using the FoxPro OleDb driver:

```
private Artist[] GetArtistsData()
{
    using (var sql = new SqlDataAccess(CONNECTION_STRING))
    {
        IEnumerable<Artist> artists = sql.Query<Artist>("select * from artists");

        if (artists == null)
            throw new ApplicationException(sql.ErrorMessage);

        return artists.ToArray();
    }
}
```

This code uses the Westwind.Utilities library and the SqlDataAccess helper. You pass in the name of a connection string that is defined as a constant:

```
private const string CONNECTION_STRING = "AlbumViewer";
```

And then defined the web.config file:

```
<connectionStrings>
  <add name="AlbumViewerFox" connectionString="Provider=vfpoledb.1;Data
Source=|DataDirectory|\;Exclusive=false;Deleted=true;Nulls=false;"
       providerName="System.Data.OleDb" />
</connectionStrings>
```

The |DataDirectory| value in the connection string evaluates to the *App_Data* folder in the Web project which is where the FoxPro sample data lives. If you have your data elsewhere, provide the full Windows path to the directory of free tables as I am doing here. If you have a Database, point at the DBC file instead.

The SqlDataAccess class has a number of methods to get data and can return query results as a DataReader, data table or as is the case here, or as I'm doing here as a list of objects. The Query<T> method maps fields of the result to matching properties of object type that is created when the query returns. The end result is an object list that we can return as a result value of the Service call.

When you now run this query you get a list of about 50 artists returned from the FoxPro table:

```
<ArrayOfArtist xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://albumviewerservice/albumservice/asmx/">
  <Artist>
    <Id>1</Id>
    <ArtistName>AC/DC</ArtistName>
    <ImageUrl>
      http://cps-static.rovicorp.com/3/JPG_400/MI0003/090/MI0003090436.jpg?partner=allrovi.com
    </ImageUrl>
    <AmazonUrl>
      http://www.amazon.com/AC-
DC/e/B000AQU2YI/?_encoding=UTF8&camp=1789&creative=390957&linkCode=ur2&qid=1412245004&sr=8-
1&tag=westwindtechn-20&linkId=SSZOE52V3EG4M4SW
    </AmazonUrl>
  </Artist>
  <Artist>
    <Id>2</Id>
    <ArtistName>Accept</ArtistName>
    <ImageUrl>
      http://cps-static.rovicorp.com/3/JPG_400/MI0001/389/MI0001389322.jpg?partner=allrovi.com
    </ImageUrl>
    <AmazonUrl>

http://www.amazon.com/Accept/e/B000APZ8S4/?_encoding=UTF8&camp=1789&creative=390957&linkCode=ur2&qid=14
12245037&sr=8-3&tag=westwindtechn-20&linkId=KM4RZR3ECUXWBJ6E
    </AmazonUrl>
  </Artist>
…
</ArrayOfArtist>
```

> **Note:**
> Your data folder must have the appropriate rights to be able to read the data files. When running in IIS this means the account the IIS Application Pool is running under. Make sure you either configuration your IIS Application pool with an account that has access or else modify your permissions so that the configure AppPool account has the access you need.

With very little code we've been able to retrieve data and return it as part of a service. Again – yay!

But I want to make it very clear that there are **lots of ways** to get the data into these objects. Using **SqlDataAccess** and the **Query()** method is just one way to do this. You can use raw ADO.NET (with a bunch more code), you can other data libraries (there are a ton of DAL packages for .NET), there's an Entity Framework driver for the FoxPro OleDb provider etc. I prefer using some of my own tools, which are available for free as NuGet packages because it's self contained and causes minimal friction in projects to return the data.

The point is there are lots of ways to accomplish the data assignment. How you do it is really inconsequential to the service discussion here – the key is that in order to return a proper result of this particular method an array of Artists has be returned somehow. I'll use the SqlDataAccess helper in this article to keep things simple.

Debugging in Visual Studio

One of the nice things about using .NET and Visual Studio is that it's very easy to debug Web applications including these Web Services. So if you have a problem in your code you can fire up the debugger.

To set a breakpoint in the .NET code find the line of code you want the debugger to stop and double click in the left margin or press F9 on the line you're interested in:

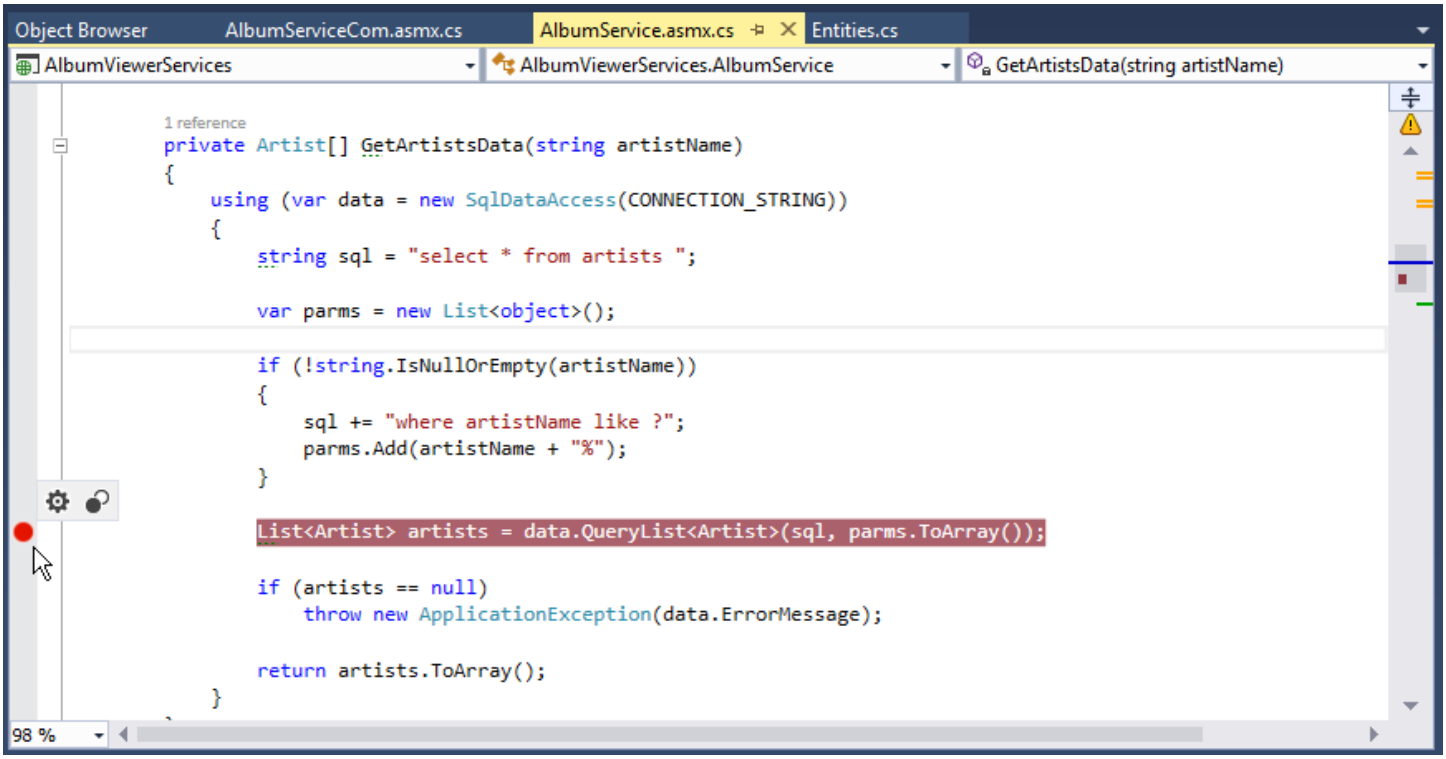


Figure 8 – Setting a breakpoint and debugging into Service code is one of the nice features of using .NET

To start debugging press F5 (or the Run button) to start the project in debug mode, then navigate to your service page and hit the service from the test page, or with SoapUI. The debugger will stop on the line with the breakpoint as you'd expect. You get a full complement of debugger features with locals and watch windows, many step options and breakpoint filters which makes it easy to trouble shoot code problems.

As mentioned earlier – if you use full IIS rather than IIS you have to run Visual Studio as an Administrator in order to be able to attach the debugger to IIS.

A more complex Query

The previous query was pretty simple. We were retrieving only a single table. Things get a little more complex if you need to deal with multiple related tables to create a hierarchical data structure. For example when we return albums we need to return nested objects for each album. An album has a single artist (1-1) and multiple tracks (1-M) associated with it and structure is represented in nested objects and collections.

As a refresher here's the album structure:

```
public class Album
{
    public int Id { get; set; }
    public int? ArtistId { get; set; }

    public string Title { get; set; }
    public string Description { get; set; }
    public int Year { get; set; }
    public string ImageUrl { get; set; }
    public string AmazonUrl { get; set; }
    public string SpotifyUrl { get; set; }
    public virtual Artist Artist { get; set; }
    public virtual Track[] Tracks { get; set; }
}
```

The code to fill this data structure requires a few additional queries. To keep the code simple I'm going to use **SqlDataAccess** again along with lazy loading to retrieve the child objects. This is not very efficient and results in a lot of database traffic, but it makes simpler code than returning and parsing a denormalized list.

Here's the code for the **GetAlbums()** and **GetAlbumsData()** methods:

```
[WebMethod]
public Album[] GetAlbums()
{
    if (App.Configuration.ProcessingMode == ProcessingModes.FoxProOleDb)
        return GetAlbumsData();

    return null;
}



public Album[] GetAlbumsData()
{
    using (var data = new SqlDataAccess(CONNECTION_STRING))
    {
        var albums = data.Query<Album>(@"select * from albums")
                         .ToList();

        foreach (var album in albums)
        {
            album.Artist = data.Find<Artist>(
                "select * from Artists where id=?", album.ArtistId);

            album.Tracks = data.Query<Track>(
                "select * from Tracks where AlbumId=?",album.Id)
                .ToList();
        }

        return albums.ToArray();
```

```
    }
}
```

The first method is the actual Service method that has the **[WebMethod]** attribute to mark it as an endpoint. The second method then does the work of retrieving the data and normally the logic in the second method would probably live in a business object.

The code retrieves a list of albums first, then goes out and loops through each album and loads the artist and track information in separate queries. Note that FoxPro uses ? for parameters placeholders in queries and these parameters are passed positionally. Unlike SQL Server and other providers that can use named parameters the Fox provider only supports positional parameters.

The result of this method is a collection of complex Album objects:

```
<ArrayOfAlbum xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://albumviewerservice/albumservice/asmx/">
  <Album>
    <Id>1</Id>
    <Title>For Those About To Rock We Salute You</Title>
    <Year>1981</Year>
    <ImageUrl>
      https://images-na.ssl-images-amazon.com/images/I/41LILmwtooL._SL250_.jpg
    </ImageUrl>
    <AmazonUrl>

http://www.amazon.com/gp/product/B00008WT5G/ref=as_li_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN
=B00008WT5G&linkCode=as2&tag=westwindtechn-20&linkId=SWZZPGAYVCI47LYK
    </AmazonUrl>
    <ArtistId>1</ArtistId>
    <Artist>
      <Id>1</Id>
      <ArtistName>AC/DC</ArtistName>
      <ImageUrl>
        http://cps-static.rovicorp.com/3/JPG_400/MI0003/090/MI0003090436.jpg?partner=allrovi.com
      </ImageUrl>
      <AmazonUrl>
        http://www.amazon.com/AC-
DC/e/B000AQU2YI/?_encoding=UTF8&camp=1789&creative=390957&linkCode=ur2&qid=1412245004&sr=8-
1&tag=westwindtechn-20&linkId=SSZOE52V3EG4M4SW
      </AmazonUrl>
    </Artist>
    <Tracks>
      <Track>
        <Id>1</Id>
        <AlbumId>1</AlbumId>
        <SongName>For Those About To Rock (We Salute You)</SongName>
        <Length>5:03</Length>
        <Bytes>11170334</Bytes>
        <UnitPrice>0.99</UnitPrice>
      </Track>
      <Track>
        <Id>6</Id>
        <AlbumId>1</AlbumId>
        <SongName>Put The Finger On You</SongName>
        <Length>3:05</Length>
        <Bytes>6713451</Bytes>
        <UnitPrice>0.99</UnitPrice>
      </Track>
      <Track>
        <Id>7</Id>
        <AlbumId>1</AlbumId>
        <SongName>Let's Get It Up</SongName>
```

```
        <Length>3:13</Length>
        <Bytes>7636561</Bytes>
        <UnitPrice>0.99</UnitPrice>
      </Track>
    </Tracks>
  </Album>
  <Album>
    <Id>2</Id>
    <Title>Balls to the Wall</Title>
    <Year>1983</Year>
    <ImageUrl>
      https://images-na.ssl-images-amazon.com/images/I/519J0xGWgaL._SL250_.jpg
    </ImageUrl>
    <AmazonUrl>

http://www.amazon.com/gp/product/B00005NNMJ/ref=as_li_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN
=B00005NNMJ&linkCode=as2&tag=westwindtechn-20&linkId=MQIHT543FNE5PNZU
    </AmazonUrl>
    <SpotifyUrl>
      https://play.spotify.com/album/2twCPCDGJjVD90GWUjA8vN
    </SpotifyUrl>
    <ArtistId>2</ArtistId>
    <Artist>
      <Id>2</Id>
      <ArtistName>Accept</ArtistName>
      <ImageUrl>
        http://cps-static.rovicorp.com/3/JPG_400/MI0001/389/MI0001389322.jpg?partner=allrovi.com
      </ImageUrl>
      <AmazonUrl>

http://www.amazon.com/Accept/e/B000APZ8S4/?_encoding=UTF8&camp=1789&creative=390957&linkCode=ur2&qid=14
12245037&sr=8-3&tag=westwindtechn-20&linkId=KM4RZR3ECUXWBJ6E
      </AmazonUrl>
    </Artist>
    <Tracks>
      <Track>
        <Id>2</Id>
        <AlbumId>2</AlbumId>
        <SongName>Balls to the Wall</SongName>
        <Length>5:02</Length>
        <Bytes>5510424</Bytes>
        <UnitPrice>0.99</UnitPrice>
      </Track>
      <Track>
        <Id>5090</Id>
        <AlbumId>2</AlbumId>
        <SongName>Fight it back</SongName>
        <Length>3:57</Length>
        <Bytes>0</Bytes>
        <UnitPrice>0.00</UnitPrice>
      </Track>
      <Track>
        <Id>5091</Id>
        <AlbumId>2</AlbumId>
        <SongName>London Leatherboys</SongName>
        <Length>3:12</Length>
        <Bytes>0</Bytes>
        <UnitPrice>0.00</UnitPrice>
      </Track>
    </Tracks>
  </Album>
…
</ArrayOfAlbum>
```

Passing Parameters

At this point we can see how to get data out of the service. Now let's see how we can pass data to the service and use it to filter our results. To do this lets add a few filter conditions to our Album query. Let's start with a very simple way to pass data – simple parameters. Let's add a string parameter to let us filter the returned artist list by artist name.

So I'm going to modify the *GetArtists()* method by adding a single parameter to it like this:

```
public Album[] GetArtists(string artistName)
```

I then also pass that parameter forward into the *GetArtistData()* method and process the parameter as part of the sql statement:

```
private Artist[] GetArtistsData(string artistName)
{
    using (var data = new SqlDataAccess(CONNECTION_STRING))
    {
        string sql = "select * from artists ";

        var parms = new List<object>();

        if (!string.IsNullOrEmpty(artistName))
        {
            sql += "where artistName like ?";
            parms.Add(artistName + "%");
        }

        List<Artist> artists = data.QueryList<Artist>(sql,parms.ToArray());

        if (artists == null)
            throw new ApplicationException(data.ErrorMessage);

        return artists.ToArray();
    }
}
```

The code accepts a parameter and then changes the SQL statement based on the parameter passed to filter the data. It also creates a parameter array so it knows how to add the parameters in the right order. It doesn't really matter here because we have a single parameter to pass, but if you had multiple parameters you have to ensure that the number of parameters matches the ? in the query. The easiest way is to add the ? to the string and the list at the same time. The query is then executed with the Sql parameters array as a parameter.

And voila we can now filter our list of artists by name partial name.

Passing Complex Parameters

Simple parameters are useful but in most WebServices pretty rare. Most of the time parameters are objects. To demonstrate a simple example, let's query our albums list and pass in an object that allows querying for multiple filters.

Let's start by creating a filter class:

```
public class AlbumFilterParms
{
    public string AlbumName { get; set; }
```

```
    public int AlbumYear { get; set; }
    public string ArtistName { get; set; }
    public string TrackName { get; set; }
}
```

Let's create another method that takes this object as an input parameter and then filters the data for each of those filters provided:

```
public Album[] GetAlbumsDataQuery(AlbumFilterParms filter)
{
    using (var data = new SqlDataAccess(CONNECTION_STRING))
    {
        string sql = "select alb.* " +
                     "      from albums alb, artists as art, tracks as trk " +
                     "      where art.id = alb.artistId and trk.albumid = alb.id ";

        var parms = new List<object>();

        if (filter != null)
        {
            if (!string.IsNullOrEmpty(filter.AlbumName))
            {
                sql += "and alb.title like ? ";
                parms.Add(filter.AlbumName + "%");
            }

            if (filter.AlbumYear > 1900)
            {
                sql += "and alb.year like ? ";
                parms.Add(filter.AlbumYear);
            }

            if (!string.IsNullOrEmpty(filter.AlbumName))
            {
                sql += "and art.ArtistName like ? ";
                parms.Add(filter.ArtistName + "%");
            }
            if (!string.IsNullOrEmpty(filter.TrackName))
            {
                sql += "and trk.SongName like ? ";
                parms.Add(filter.TrackName + "%");
            }
        }

        var albums = data.QueryList<Album>(sql, parms.ToArray());

        foreach (var album in albums)
        {
            album.Artist = data.Find<Artist>(
                "select * from Artists where id=?", album.ArtistId);

            album.Tracks = data.Query<Track>(
                "select * from Tracks where AlbumId=?", album.Id)
                .ToList();
        }

        return albums.ToArray();
    }
}
```

Because we are now passing a complex object as a parameter the simple test page no longer works to test our service. In order to test this service method we have to use a SOAP client or SoapUI. In Soap UI here's what a request looks like:

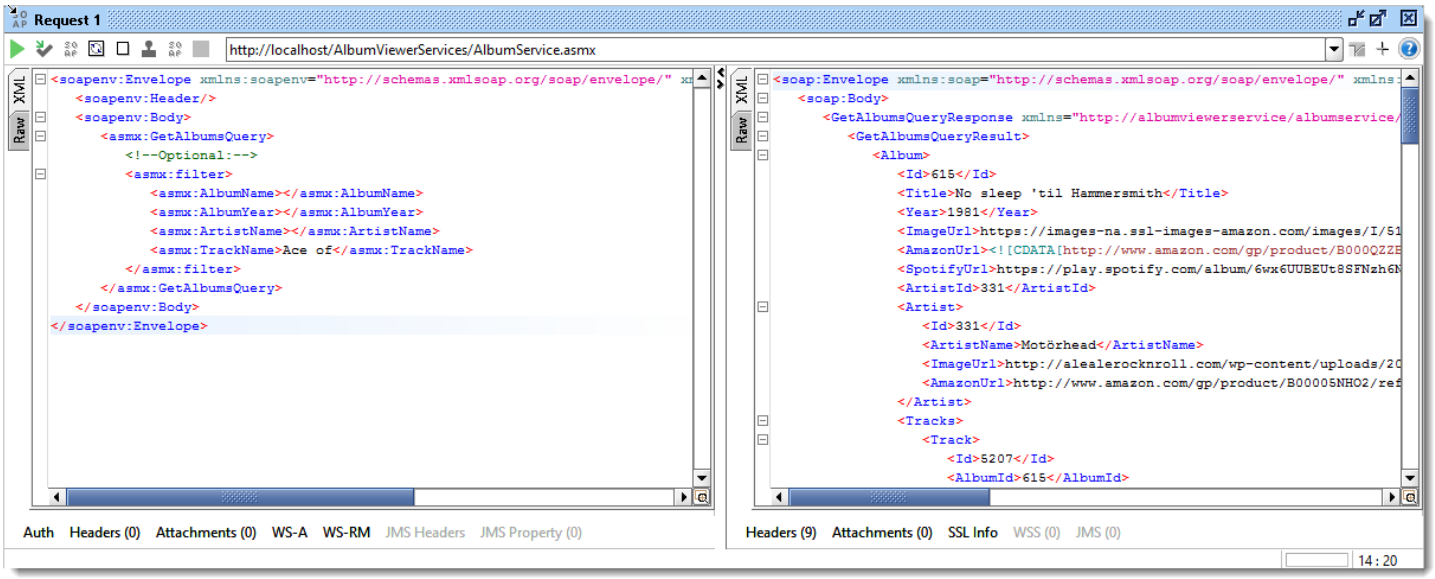


Figure 9 – To test complex object inputs we have to use a full Soap client to test requests rather than the ASMX test page. SoapUI makes it easy.

Notice that all of the object parameters are optional. If I remove them the request still works. In the example I specify a track name search, which in this case finds two albums to display.

Returning a single Object

In order to have a simple client example, let's add another method to return a single abum from the service. We'll pass in an album name and the service returns a single instance of an album in all of its hierarchical glory. Here are the two methods:

```
[WebMethod]
public Album GetAlbum(string albumName)
{
    if (App.Configuration.ProcessingMode == ProcessingModes.FoxProOleDb)
        return GetAlbumData(albumName);

    return null;
}


public Album GetAlbumData(string albumName)
{
    using (var data = new SqlDataAccess(CONNECTION_STRING))
    {

        string sql = "select * from albums where title like ?";

        var album = data.Find<Album>(sql, albumName + "%");
        if (album == null)
            throw new ArgumentException("Album not found: " + data.ErrorMessage);

        album.Artist = data.Find<Artist>("select * from artists where id=?", album.ArtistId);
        album.Tracks = data.QueryList<Track>("select * from tracks where albumid=?", album.Id);

        return album;
    }
```

```
}
```

The result for this is a single album object with Tracks and an Artist associated. We'll use this method as our first example when calling our service from FoxPro.

Consuming the Web Service From FoxPro

In order to use this service in FoxPro I'm going to create another .NET project that acts as a service proxy for the Web project. We will then call that .NET component from FoxPro to make our service calls.

Creating a .NET Assembly as a Proxy

The first step is to create .NET service proxy. There are a number of ways you can do this and since this is a SOAP 1.x service I'm going to use the WSDL.exe based client to handle this which is the easiest way to import a Web service.

First create a new .NET Class Library Project:

- On the Solution node in the project click Add | New Project
- Select Visual C# | Class Library
- Name it AlbumViewerProxy

This will create a new project in the existing solution. If you're creating a new project just choose ClassLibrary as you top level project when you start out.

Once the project has been created go the References node, right click and select *Add Service Reference.*

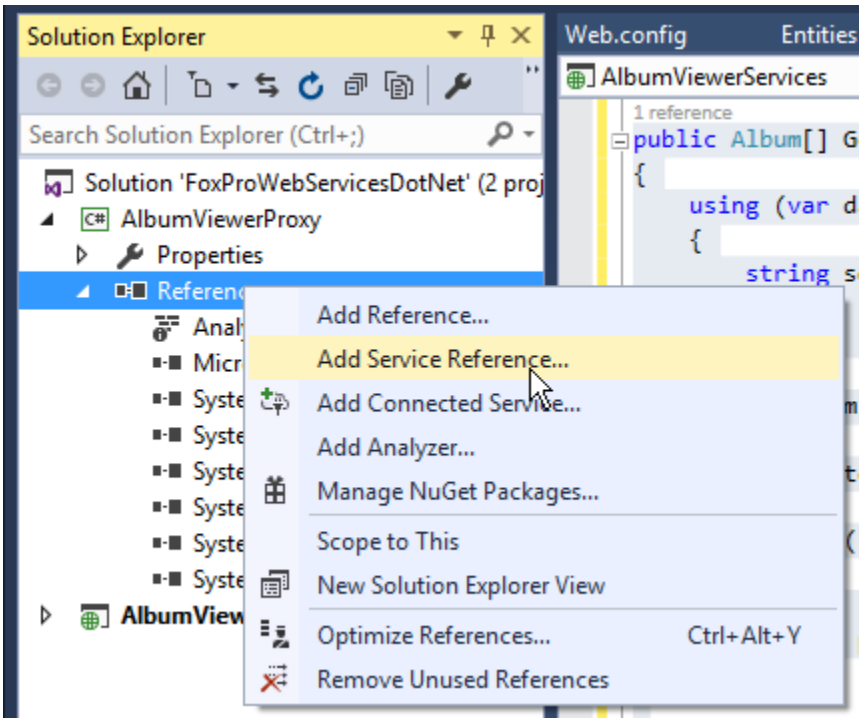


Figure 10 – Adding a Service reference to add a Web Service to a Visual Studio Project

Because WSDL based services are old they are a little tricky to find. You have to click through two dialogs to get to the actual Service import dialog.

On the first two dialogs just click the Add Web Reference buttons on the bottom:

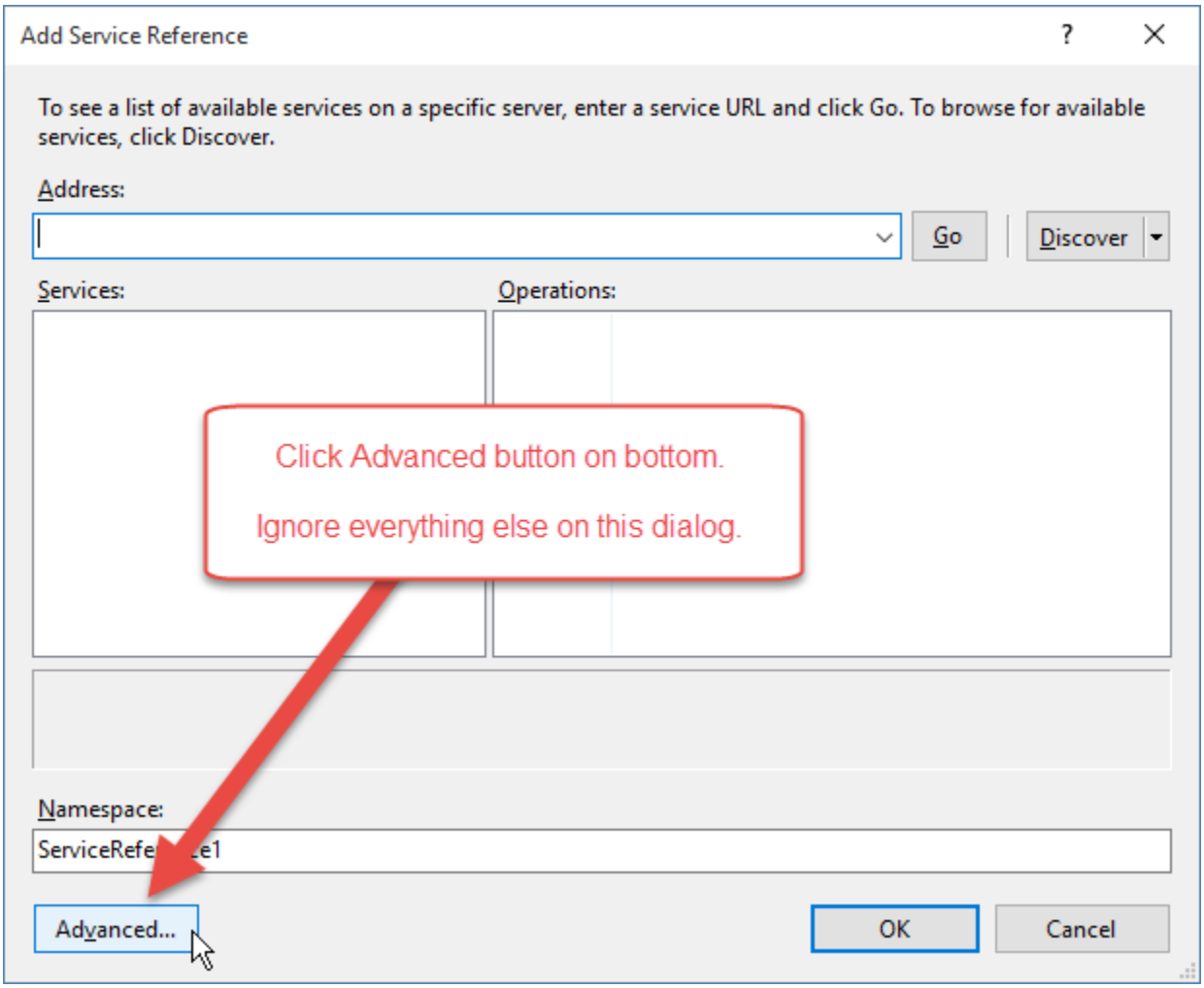


Figure 11 – Skip over the standard dialogs to get to the 'classic' Web services

You don't need to fill in anything because this dialog and the next are simply ignored.

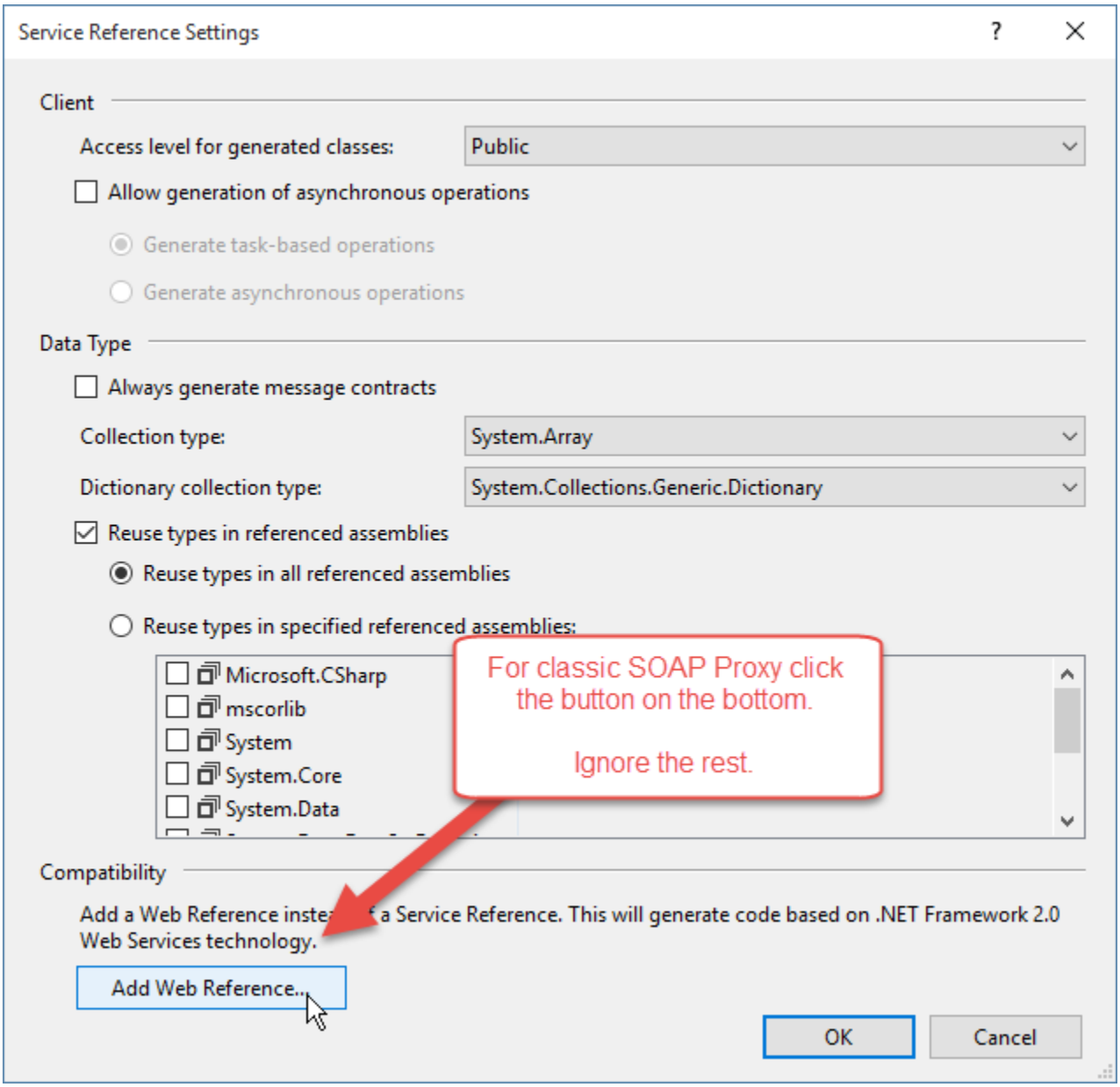


Figure 12 – More skipping

The final dialog that pops up is the one where we actually can specify the service URL by providing a link to the WSDL.

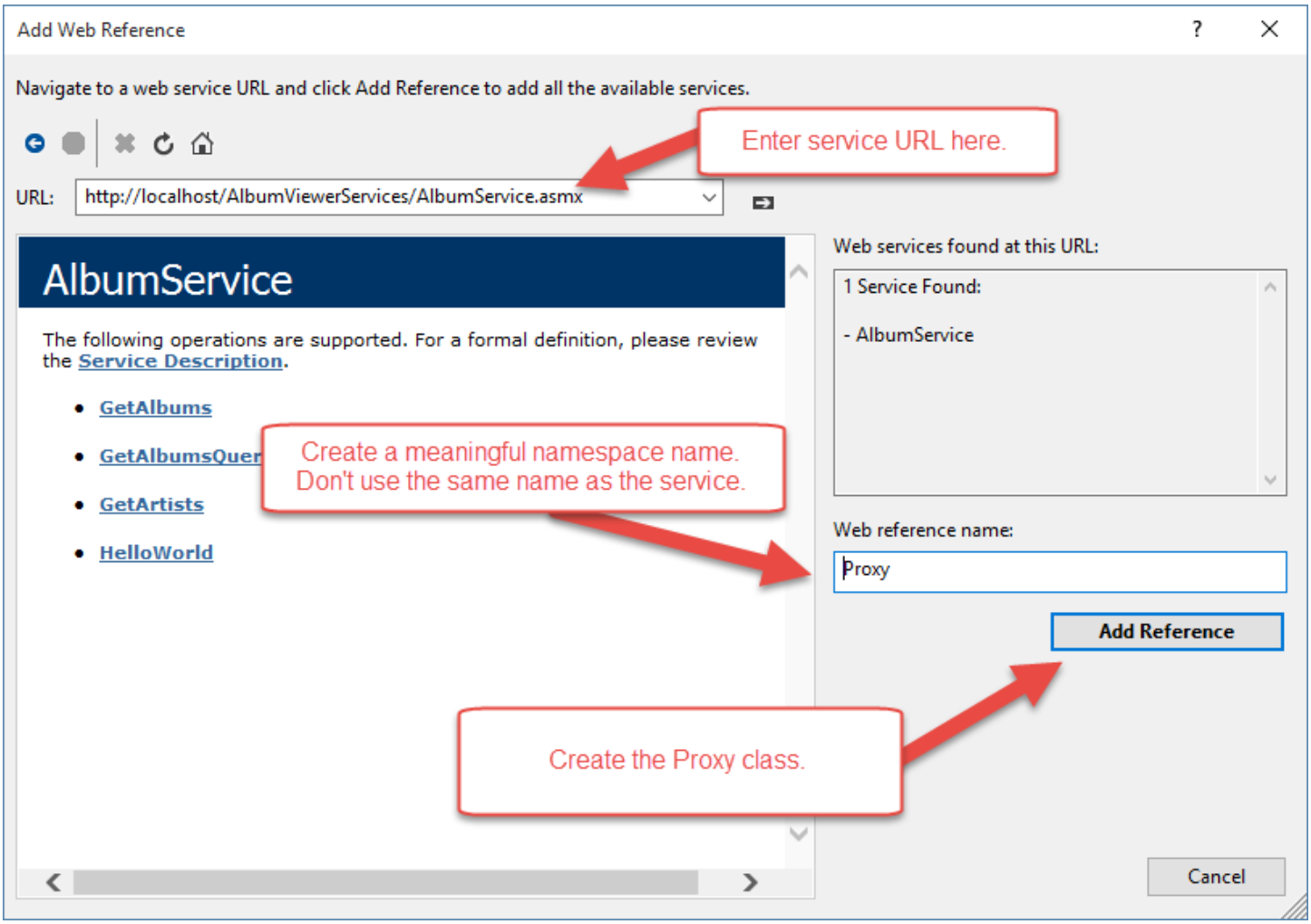


Figure 13 – Adding a classic Web Service reference. Make sure you use a sensible Reference name which translates into the namespace used. Don't use the same name as the service!

Once you've done this Visual Studio has added a new proxy class into your project. Compile the project first then you can look at the actual service proxy that was created in the Visual Studio Object browser to see what was generated.

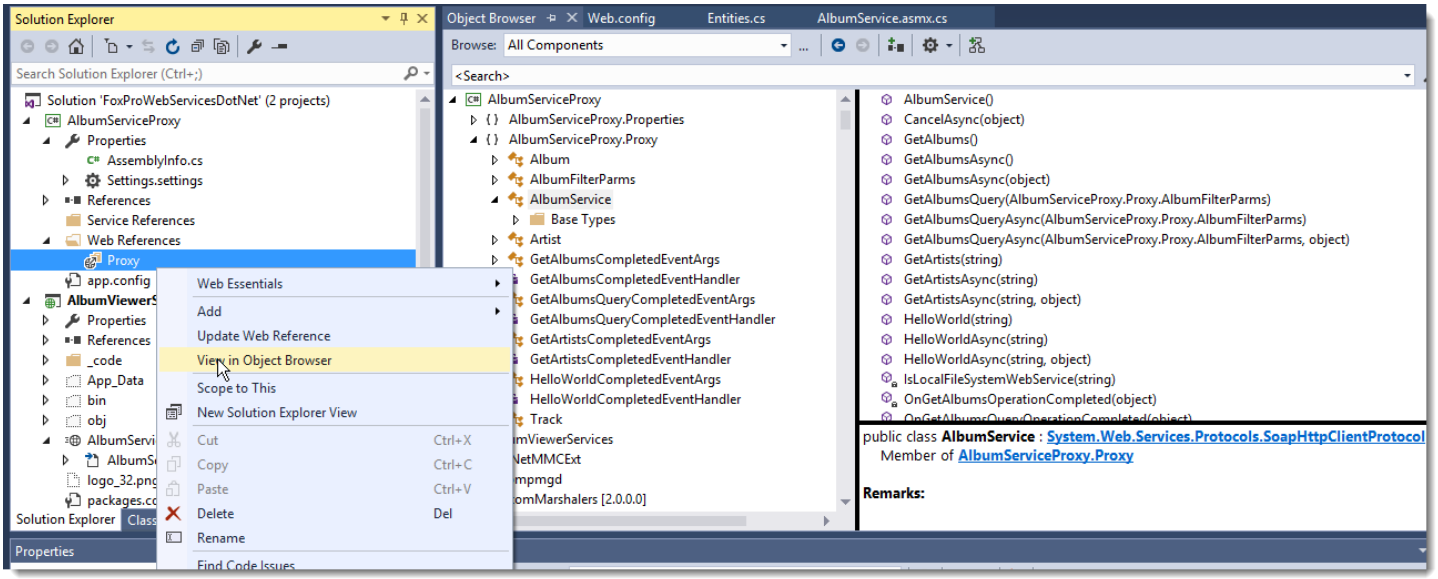


Figure 14 – The object browser shows you the generated .NET classes. It also provides you the info you need to instantiate these classes from FoxPro, namely the full namespace and class name.

The proxy import generates a service class (highlighted) as well as all of the related message objects that the service uses – so there are instances of Album, Artist, Track, and the query filter class in addition to the actual main service class.
Remember the Object Browser in Visual Studio (or Reflector which I'll get to shortly) – you can use it to discover the exact signatures required to create instances of these .NET classes.

Generate a Proxy without Visual Studio

You can also remove Visual Studio from this equation if all you want to do is generate the proxy and create the .NET DLL. Frankly there's not much value added in Visual Studio unless you plan on extending the functionality of the service or add it to other features you want to call.

To use the command line, open a Windows Command prompt and go to the *\Fox\Tools* folder in the samples for this article. You'll find a CreateDll.bat file in this folder. You can run it with:

CreateDll.bat http://localhost/AlbumViewerServices/AlbumService.asmx Proxy AlbumService

You pass 3 parameters:

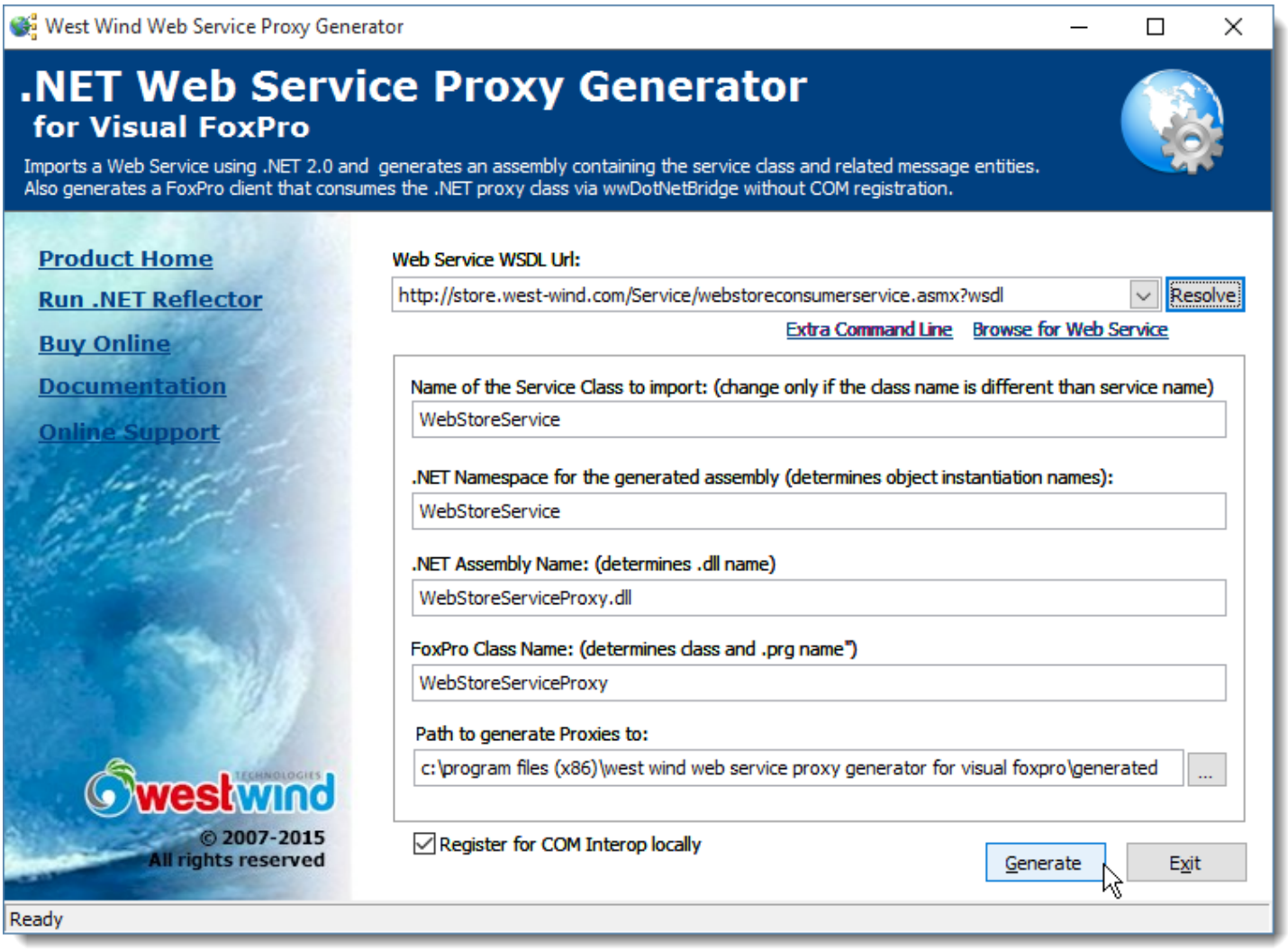
- The URL to the WSDL document
- The namespace of the generated class
- The name of the generated DLL (no extension)

This creates an AlbumService.dll that contains the generated proxy and you get a similar proxy to the one created in Visual Studio. Even better this proxy is generated without all the async methods as it uses a configuration file to only export what's needed. Once you're done copy the generated .dll file into your FoxPro project directory.

Using the West Wind WSDL Proxy Generator

Yet another option to generate these .NET proxies as well as a FoxPro client class to access the proxy's service methods is the West Wind Web Service Proxy Generator. It uses the same tooling described above to generate the .NET client for you without requiring Visual Studio, but then also creates a FoxPro class that calls the all the service methods. So you don't have to write any code to perform the actual service mappings that I'll describe in the next section.

The nice thing about this is that the entire process is automated as it keeps the Web Service, the .NET proxy and the FoxPro client code in sync in one step. The generated FoxPro and .NET proxies also add error handling to capture service exceptions, handling authentication and adding SOAP headers which is beyond the scope of this article.



It's a quick and easy solution to creating Web Service clients for those that want to just get going quickly that automates a lot of what is discussed in the next section.

This tool does nothing that you can't manually do yourself – it just automates the process and provides some of the more advanced functionality in a pre-packaged client for you. Much of the content in this article is based on the experiences on building and using this tool with various clients calling a huge variety of Web Services.

Watch out for Service Changes

Note for any of these solutions require you to reload the service references when the service changes. So if you're building both the client and the server at the same time, it's critical that your client proxies are updated to reflect any changes in the server interface.

We now have a .NET service – how do we call this from FoxPro?

Using wwDotnetBridge for COM Interop

Although it's possible to use plain COM interop to call a .NET component that has been exported to COM, I'd highly recommend you use wwDotnetBridge instead. COM Interop has many problems when it comes to certain .NET types that are returned from services. Specifically dealing with arrays, numeric data types, enumerations and value types is either difficult or not supported at all with plain COM interop. Additionally in order to use COM interop you have to register your components using non-standard tooling that is not readily available.

wwDotnetBridge bypasses many of these issues by allowing direct access to .NET components without COM registration, and a ton of helpers that simplify working with problem .NET types. wwDotnetBridge is open source and available for free or if you are using West Wind Web Connection or West Wind Client Tools it's part of those packages as well (with some additional features). You can find out more here:

- **wwDotnetBridge on GitHub**

In order to use the generated .NET class we have to reference it from FoxPro code. I recommend that you change the path of the compiled output in .NET into your FoxPro code directory (or a bin folder below it).

To do this for Visual Studio:

- Right Click on the Project node
- Select Properties
- Go to the Build Tab
- Note the Configuration (Debug or Release)
- Change the folder to your application folder

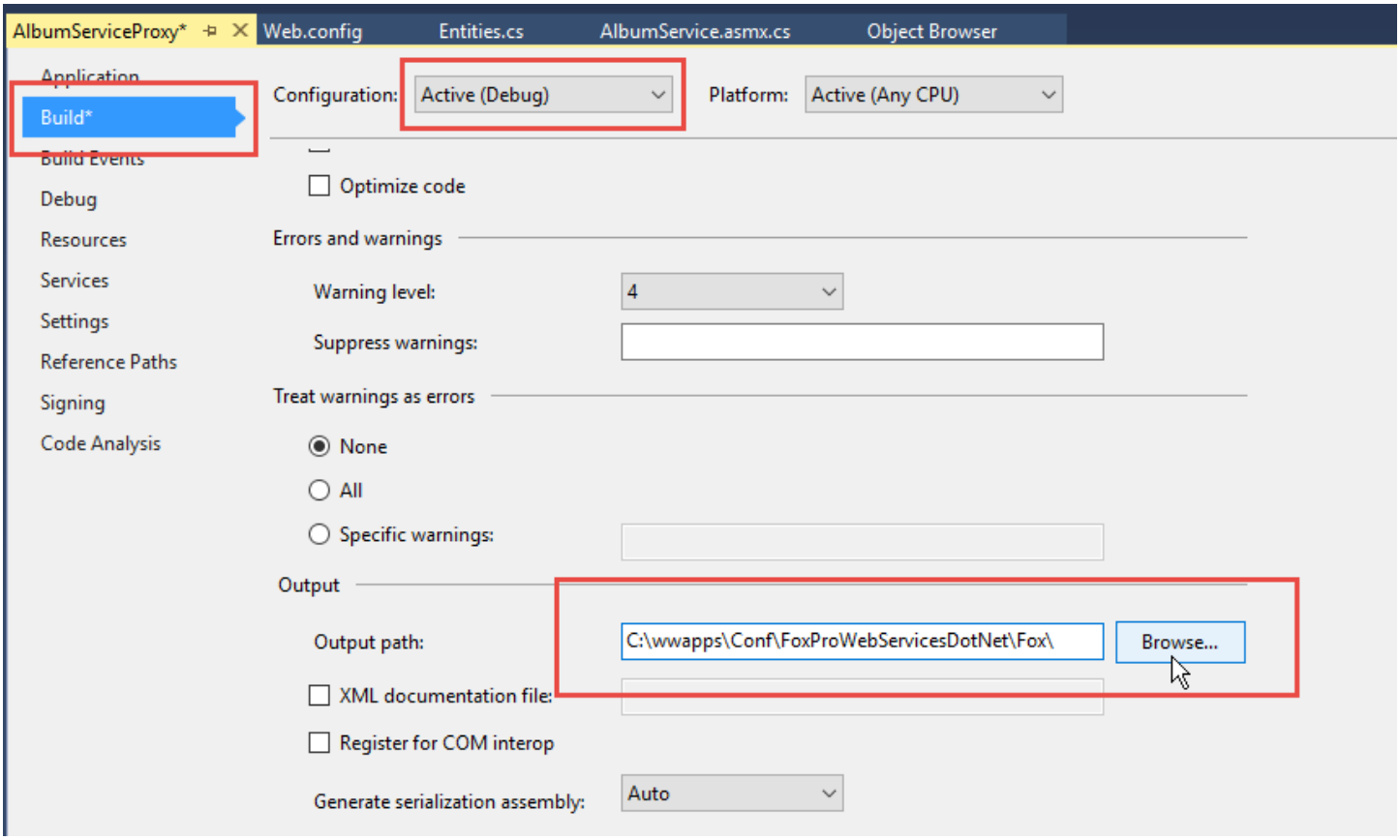


Figure 15 – It's a good idea to generate your .NET proxy DLL to the FoxPro application folder for both Debug and Release builds.

While you can easily reference the .NET dll in its original folder, it's better to have it in your application folder since you will need to have it there when you deploy your application. Note in my case this path can be shortcut to ..\Fox\

Now we can use the service from FoxPro. Let's start with the *GetAlbum()* method:

```
do wwDotNetBridge
```

```
LOCAL loBridge as wwDotNetBridge
loBridge = CreateObject("wwDotNetBridge","V4")

IF !loBridge.LoadAssembly("AlbumServiceProxy.dll")
    ? "Invalid library..." + loBridge.cErrorMsg
    return
ENDIF

LOCAL loService as AlbumServiceProxy.Proxy.AlbumService
loService =
loBridge.CreateInstance("AlbumServiceProxy.Proxy.AlbumService")

*** Always check for errors!
IF ISNULL(loService)
   ? "Unable to create service: " + loService.cErrorMsg
   RETURN
ENDIF

loAlbum =  loService.GetAlbum("Ace of Spades")

*** Always check for errors
IF ISNULL(loAlbum)
  ? "Couldn't get item: " + loService.cErrorMsg
  RETURN
ENDIF

? loAlbum.Title
? "by " + loAlbum.Artist.ArtistName
? loAlbum.Descriptio
? loAlbum.Year
```

This code calls the Web Service and retrieves a single instance of an album and displays some information about it.

The steps to using wwDotnetBridge are:

1. Load the library
2. Load the .NET Assembly you want to access
3. Create an instance of the .NET type
4. Call methods, access and set properties

To create the instance just create wwDotnetBridge and pass an optional parameter of the .NET runtime you want to use (V2 or V4 – defaults to the highest installed). Note that you can set the .NET Runtime only the first time wwDotnetBridge is invoked. After that the same runtime is used regardless of the parameter. I recommend you create a dummy instance in your startup PRG to force the runtime to be loaded under your control, so you can enforce which version loads throughout your application consistently.

Next you call LoadAssembly. You can load any .NET dll this way simply by referencing it by its path. You can also reference GAC components by using the fully qualified assembly name (a long string). When assemblies are loaded all of their dependent assemblies are also loaded automatically as needed (JIT), but

you have to ensure all the dependencies are available. LoadAssembly has to be called only once but it won't fail if you call it multiple times.

Once the assembly is loaded you can start calling methods on the service. All the [WebMethod] endpoint methods are available to you to call and access from the proxy. In the sample I'm calling the GetAlbum() method which returns a single album object instance.

If the result comes back with an instance you can simply access properties of that object. So I can look at the Title and Description and even the child Artist object.

A little later I'll show you how can abstract a lot of this noise away when you call services that have many methods – you don't want to repeatedly call all of this setup code, but you rather abstract the service reference into a FoxPro class and then wrapper the actual methods you are calling. More on this later.

Changing the Service Url

One important thing that you might have to do when you call a service is test the service against a test server and then switch the service to live production URL which requires changing the service endpoint URL. The WSDL document that you generate the service against has a service URL embedded in it and that's the URL that is used by default.

If you need to switch to another URL you can specify it explicitly by using the URL property of the service proxy. Unfortunately you can't do:

```
loService.Url = "http://localhost/production/albumservice.asmx"
```

Rather you have to use wwDotnetBridge to access the URL property on the service instance:

```
loBridge.SetProperty(loService,"Url","http://localhost/production/albumservice.asmx")
```

Ugly, but required because the Url property on the service class is inherited and COM Interop doesn't pick up inherited members from abstract base classes. It's another one of those COM Interop quirks that wwDotnetBridge allows you work around.

Arrays and Collections

You may notice that I artfully neglected to access the Tracks property. That's because the Tracks property is an array property and in order to effectively deal with arrays a little more work is required. While you can access array elements directly it's better if you use some of the wwDotnetBridge helpers to provide an array wrapper. The wrapper makes it easy to iterate over the array, update elements and then also send them back up to the server.
Here's the code to list the tracks using wwDotnetBridge and indirect referencing:

```
*** Get tracks Array as a COM Collection
loTracks = loBridge.GetProperty(loAlbum,"Tracks")

IF loTracks.Count > 0
     FOR lnX = 0 TO loTracks.Count - 1
          loTrack = loTracks.Item(lnX)
          ? "    " + loTrack.SongName + " " + loTrack.Length
     ENDFO
ENDIF
```

Notice the call to *GetProperty()* to convert the .NET array into something that FoxPro can deal with more easily. This returns a *ComArray object* that wraps a .NET array and leaves that array in .NET – it is never marshalled into FoxPro directly. You can retrieve items in the array, update elements and add new ones, but the array instance never passes directly into FoxPro. This allows two-way editing and updates safely.

Once you have this *ComArray* instance you can ask for its Count property, then loop through each of its *.Item()* methods retrieve the individual items.

wwDotnetBridge auto converts a number of types automatically when using *GetProperty(), SetProperty()* and *InvokeMethod()* to indirectly access .NET objects. If you have problems working with properties or methods directly try using these indirect methods instead.

Returning Collections From the Service

So far so good. Let's try calling the other two methods. For the most part the code for these methods will be similar. Here's the code for GetArtists() minus the initial setup code:

```
loArtists =  loBridge.InvokeMethod(loService,"GetArtists","") && all


FOR lnX = 0 to loArtists.Count - 1
     loArtist = loArtists.Item(lnx)
     ? "   " + loArtist.ArtistName
ENDFOR
```

GetArtists() returns a collection of artist objects so here we need to right away use *.InvokeMethod()* to retrieve the result as a *ComArray* object. We can then loop through use the *.Item()* method to retrieve the individual artists and display them.

To get Albums works very similarily, but we can do some nested enumeration of the tracks:

```
loAlbums = loBridge.InvokeMethod(loService,"GetAlbums")

*** Always check for errors
IF ISNULL(loAlbums)
  ? "Couldn't get item: " + loService.cErrorMsg
  RETURN
ENDIF

FOR lnX = 0 TO loAlbums.Count - 1

    loAlbum = loAlbums.Item(lnx)
     ? loAlbum.Title
     ? "by " + loAlbum.Artist.ArtistName
     ? loAlbum.Descriptio
     ? loAlbum.Year

     *** Get tracks Array as a COM Collection
     loTracks = loBridge.GetProperty(loAlbum,"Tracks")

     IF loTracks.Count > 0
          FOR lnY = 0 TO loTracks.Count - 1
```

```
                loTrack = loTracks.Item(lnY)
                ? "    " + loTrack.SongName + " " + loTrack.Length
            ENDFO
        ENDIF
ENDFOR
```

At this point you can see it's pretty straight forward to retrieve and display data from a service.

Passing Data to a Service

Next up we need to pass some data to the service. Let's look at the GetAlbumsQuery() method which receives the filter class as a parameter. In order to pass an object to .NET we have to ensure that the object passed is of the proper type – it has to be the **EXACT** type that is specified in the parameter signature, which means we have to create a .NET object and pass it to .NET as a parameter.

The first thing required is the exact .NET typename we need to pass and to do that we can use the Object Browser in Visual Studio or a tool like .NET Reflector (in the Tools directory). I'm going to look up the AlbumFilterParms class and look at the type signature in the Object Browser in VS:

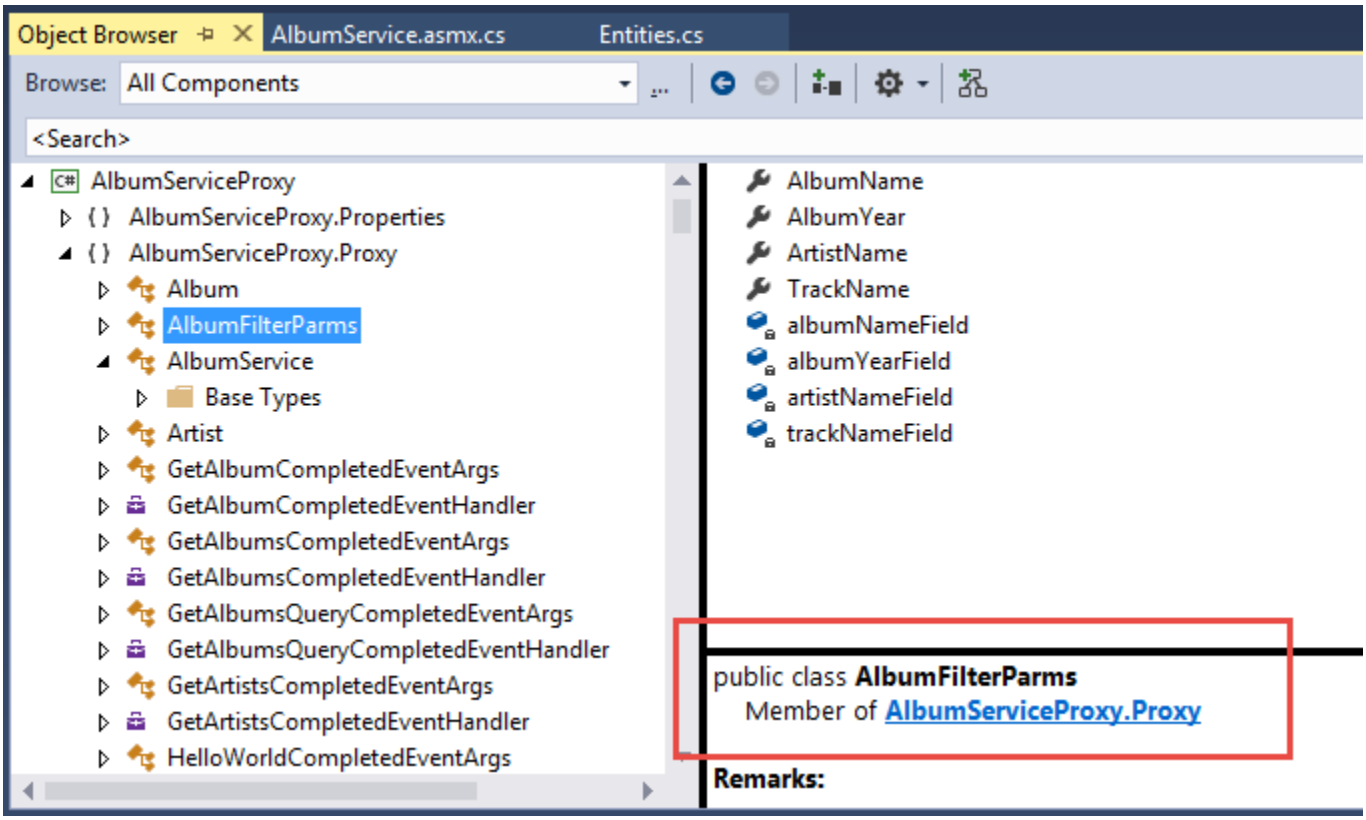


Figure 16 – Use the Object Browser (or Reflector) to find out the example name of a .NET type to instantiate. A full type name is *namespace.classname.*

If you look at the details of the class you see the name of the class and the namespace (Member of which is a little misleading). The full type name is **AlbumServiceProxy.Proxy.AlbumFilterParms** which is the namespace plus the classname combined by a dot.

The object browser is also very useful in showing what properties and methods are available on objects and what their exact type signatures are. It's important to know what you need to pass to each method for example.

With that we can now create an instance of this type, set its properties and pass it to the *GetAlbumsQuery()* method.

```
loFilter = loBridge.CreateInstance(
                      "AlbumServiceProxy.Proxy.AlbumFilterParms")
loFilter.AlbumName = "Ace of Spades"

loAlbums = loBridge.InvokeMethod(loService,"GetAlbumsQuery",loFilter)

*** Always check for errors
IF ISNULL(loAlbums)
  ? "Couldn't get item: " + loService.cErrorMsg
  RETURN
ENDIF

FOR lnX = 0 TO loAlbums.Count - 1

    loAlbum = loAlbums.Item(lnx)
     ? loAlbum.Title
     ? "by " + loAlbum.Artist.ArtistName
     ? loAlbum.Descriptio
     ? loAlbum.Year

     *** Get tracks Array as a COM Collection
     loTracks = loBridge.GetProperty(loAlbum,"Tracks")

     IF loTracks.Count > 0
          FOR lnY = 0 TO loTracks.Count - 1
               loTrack = loTracks.Item(lnY)
               ? "   " + loTrack.SongName + " " + loTrack.Length
          ENDFO
     ENDIF
ENDFOR
```

The key in this code is the loBridge.CreateInstance() call that effectively creates a .NET object and makes it available to your FoxPro code. You set properties and then pass that object to .NET which recognizes it as the proper type for the filter object. After that the code is the same as before.

So now we've come full circle – we've create services from FoxPro data and returned those results as SOAP results from the service. We can capture those results in FoxPro using .NET as a bridge to make the service calls and we can push data to .NET from FoxPro by using .NET objects created in FoxPro.

Life is good. But – there is more!

Using FoxPro COM Objects on the Server to return Data

To this point we've used OleDb and .NET code to manage our data access. This works and it's a good solution if your data needs are relatively simple and you're willing to write .NET data access and business logic code to access your data.

If you have existing complex business logic in FoxPro that you want to share, then you might want to use VFP COM objects from your Web service to provide the data layer that you can drive from the Web

service. This allows you to reuse existing business logic and offload the main coding into your FoxPro COM server rather than writing .NET code.

Keep in mind though that doing COM development can be challenging as it's not easy to debug COM servers at runtime and you have to deal with loading/unloading of servers in order to recompile and update code. There are also threading issues you need to consider for Web services – unlike ASP.NET WebForms ASP.NET Web Services do not support STA apartment threads out of the box. Some custom configuration is required to ensure COM servers can execute safely.

A Word about IIS Configuration

In order to run FoxPro COM objects in IIS make sure that you configure the following things:

- **The Application Pool Identity for the Virtual or Web Site**
  Make sure the identity is set to a user account that has rights to access your data and configuration files, network rights etc. This required both for COM interop as well as OleDb data access.

- **Set the Application Pool to 32 bit mode**
  FoxPro DLL COM objects are 32 bit, and if you call them from a 64 bit .NET application, they will fail. The error will be in-descript so be pro-active and make sure your Application pool that hosts your VFP components is 32 bit.

Creating a COM Server

Creating a COM Server in FoxPro is pretty simple – you create a FoxPro class and you add an OlePublic keyword to it. Compile to an MTDLL and voila you have a DLL COM server you can use and call from .NET.

However, the reality is that you have to take on how you create your COM objects. COM objects invoked from IIS (or any service for that matter) will run out of arbitrary places, use specific security contexts and will load and reload your components on every hit. All of these things require some attention. Take a look at the related ASP.NET Interop document that discusses these issues in more detail and they equally apply to any COM servers you create for Web or other services.

To demonstrate lets create a FoxPro COM server that mimics some of the behaviors we created previously for retrieving albums and artists and then also updates some data. First let's create the COM object in FoxPro by creating a FoxPro class:

```
SET PROCEDURE TO aspbase.prg ADDITIVE


************************************************************
DEFINE CLASS AlbumServer as AspBase OLEPUBLIC
************************************************************

FUNCTION Init()

DODEFAULT()
SET PATH TO THIS.ApplicationPath + "..\AlbumViewerServices\App_data\;" + ;
             THIS.ApplicationPath + "weblog\;"

ENDFUNC
*   Init

ENDDEFINE
```

When I create a COM server for use in ASP.NET or any server side application I always use the *AspBase* base class. AspBase is a custom base class that inherits from Session and provides a few base services that helps set up the COM object by setting the path to the DLL folder, setting the environment and a few other things. Here's what the Init() of AspBase looks like:

```
************************************************************
DEFINE CLASS AspBase AS Session
************************************************************

*** 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

*** Stock Properties
*** Set Environment
************************************************************************
* 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( JustPath(Application.ServerName) )

*** 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 ) ADDITIVE

ENDFUNC

…

ENDDEFINE
```

AspBase sets your environment, adds the DLL folder to your PATH, sets the c*ApplicationPath* property so you can set paths to other related paths like your data (if it's relative) and sets up some helper functions. This is very important! Services run in the system context so if your application depends on relative paths to find data or configuration files you'll want to use this cApplicationPath property.

There's a ComHelper object that can be used to create instances of FoxPro objects, Eval and Execute commands, and convert cursors to xml and collections and back. It's a small class, but it does all the things that you should do on every COM object anyway so it's a good idea to inherit from it or something like it. To use the base class simple implement Init() on your COM object and call DoDefault().

Accessing Data and Passing it to .NET

With the administrative stuff out of the way let's create a GetArtists() method. We'll pass a single parameter for the artist name to filter the data on optionally.

```
FUNCTION GetArtists(lcName)

IF EMPTY(lcName)
  lcName = ""
ENDIF
lcName = lcName + "%"

SELECT * FROM Artists ;
   WHERE ArtistName like lcName ;
   ORDER BY ArtistName ;
   INTO CURSOR TQuery

loCol = CursorToCollection("TQuery")

USE IN TQuery

RETURN loCol
```

Short and sweet, right? The code runs a query to get the Artists and then calls a helper function (from AspBase.prg) to convert the cursor into a Collection object, which is then returned from the COM server.

When building COM objects always, always test your code locally first to make sure it works before throwing it into COM:

```
loServer = CREATEOBJECT("AlbumServer")
loArtists = loServer.GetArtists("")
? loArtists.Count
```

This should return 50 or so artists. Since it works, let's create a project, add the AlbumServer.prg file to the project and then compile the project to an MTDLL:

```
BUILD MTDLL albums FROM albums recompile
```

Test the server again, this time as a COM object:

```
loServer = CREATEOBJECT("Albums.AlbumServer")
loArtists = loServer.GetArtists("")
? loArtists.Count
? loArtists.Item(2).ArtistName
```

If that's all still good we can now call this from our service. To do this I'm going to create a new Service and call it AlbumServiceCom.asmx. Follow the steps from earlier by adding a new ASMX service to the Web project.

Open the AlbumServiceCom.asmx.cs file and add the following code to retrieve and use the COM object and result data:

```
[WebService(Namespace = "http://albumviewerservice/albumservicecom/asmx/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
[System.ComponentModel.ToolboxItem(false)]
public class AlbumServiceCom : System.Web.Services.WebService
{

    [WebMethod]
    public Artist[] GetArtists(string artistName)
    {
        dynamic fox = CreateObject();

        dynamic artists = fox.GetArtists(artistName);
        var artistList = new List<Artist>();

        // IMPORTANT: Fox collections are 1 based!
        for (int i = 1; i <= artists.Count; i++)
        {
            dynamic foxArtist = artists.item(i);

            var artist = new Artist();

            artist.Id = (int) foxArtist.Id;
            artist.ArtistName = foxArtist.ArtistName;
            artist.Descriptio = foxArtist.Descriptio;

            artistList.Add(artist);
        }

        return artistList.ToArray();
    }

    private dynamic CreateObject(string ProgId = "Albums.AlbumServer")
    {
            Type type = Type.GetTypeFromProgID(ProgId, true);
            return  Activator.CreateInstance(type);
    }

}
```

As before remember to assign a namespace Uri to the service declaration. To create a new endpoint we just create a new [WebMethod] and add our code for GetArtists().

The code starts by creating an instance of our COM object using Reflection. Reflection is .NET's meta-data API that allows dynamic creation and invocation of types and members. One of the features is to map COM objects to a .NET _ComObject type and this type can then be instantiated.

Due to C#'s strongly typed language aspects in the past this was difficult to do – you had to map the COM object to a .NET object, or use raw Reflection APIs to access the COM object which was very verbose. However since .NET 4.0 .NET has a **dynamic** data type that lets us access the COM object without using Reflection or mapping it to a .NET type first. Instead we can just reference properties and method directly. Dynamic uses runtime member discovery so unlike normal types there's no Intellisense or type checking. If you misspell a property or method or you misassign a value to the wrong type you'll get runtime errors.

Using the dynamic COM instance we can call the **.GetArtists()** method to retrieve the Fox collection of artists, then loop through it, pick out the individual Artist instance and then assign the value to a manually created and mapped instance of the .NET Artist class. It's important to note that FoxPro Collections – unlike .NET collections – are 1 based so make sure the loop counter starts at 1 not 0. Been there done that, puzzled over it because the error message just tells you there's a null reference…

Similar to the OleDb code we wrote, we're essentially mapping the FoxPro class values to the .NET object, then adding the .NET object to a list which at the end is returned from the ASMX service method. The result of this method then is the same as what we saw with the OleDb methods.

A little more Complexity

To demonstrate a more complex object structure let's also implement the GetAlbums() method which returns a collection of albums with the associated artist and tracks. This is a bit more involved both on the FoxPro end and the .NET side.

Here's the FoxPro code for the COM object:

```
FUNCTION GetAlbums(lcTitle)

IF EMPTY(lcTitle)
 lcTitle = ""
ENDIF
lcTitle = lcTitle + "%"

SELECT  * FROM Albums ;
   WHERE Title like lcTitle ;
   ORDER BY TITLE ;
   INTO CURSOR TAlbums

loAlbums = CREATEOBJECT("Collection")
SCAN
   LOCAL loAlbum
   loAlbum = ""
   SCATTER NAME loAlbum Memo
   loAlbums.Add(loAlbum)
ENDSCAN

FOR EACH loAlbum IN loAlbums FOXOBJECT
   SELECT * FROM artists WHERE Id = loAlbum.ArtistId INTO CURSOR TArtist

   SCATTER NAME loArtist MEMO
   ADDPROPERTY(loAlbum,"Artist",loArtist)

   SELECT * FROM Tracks WHERE albumid = loAlbum.Id INTO CURSOR TTracks
   ADDPROPERTY(loAlbum,"Tracks",CursorToCollection("TTracks"))
ENDFOR

USE IN TAlbums
USE IN TTracks
```

```
USE IN TArtist

RETURN loAlbums
```

This code uses a two step process: It first retrieves the list of matching albums, and then loops through that collection and adds new properties for Artist and Tracks filled by the results of additional queries using SCATTER NAME to fill the Artist and CursorToCollection() to attach the tracks. Again – not very efficient but very straightforward code.

As always test your code first before loading it into IIS - I did ☺…

Updating Code means Recompiling your DLL and Restarting IIS

You'll need to recompile your COM server in order to access this code in .NET. The COM server likely is still loaded in your application in IIS, so you'll need to unload it first:

```
RUN /n4 IISRESET
```

Hint: I have this mapped to an IIS shortcut in _foxcode

Then:

```
BUILD MTDLL albums FROM albums RECOMPILE
```

Now switch back to Visual Studio and add the following code to the AlbumViewerCom service implementation:

```
[WebMethod]
public Album[] GetAlbums(string albumName)
{
    dynamic fox = CreateObject();
    dynamic albums = fox.GetAlbums(albumName);
    var albumList = new List<Album>();

    // IMPORTANT: Fox collections are 1 based!
    for (int i = 1; i <= albums.Count; i++)
    {
        dynamic foxAlbum = albums.item(i);

        var album = new Album();

        album.Id = (int)foxAlbum.Id;
        album.Title = foxAlbum.Title;
        album.Descriptio = foxAlbum.Descriptio;
        album.ArtistId = (int) foxAlbum.ArtistId;

        dynamic foxTracks = foxAlbum.Tracks;
        dynamic foxArtist = foxAlbum.Artist;

        album.Artist = new Artist();
        album.Artist.Id = (int) foxArtist.Id;
        album.Artist.ArtistName = foxArtist.ArtistName;

        album.Tracks = new List<Track>();

        // Fox collection is 1 based
        for (int j = 1; j <= foxTracks.Count; j++)
        {
            dynamic foxTrack = foxTracks.Item(j);
            var track = new Track();
```

```
            track.Id = (int)foxTrack.Id;
            track.SongName = foxTrack.SongName;
            track.Length = foxTrack.Length;

            album.Tracks.Add(track);
        }

        albumList.Add(album);
    }

    return albumList.ToArray();
}
```

This code uses the same principles used in *GetArtists()* but there's a bit extra manipulation for dealing with the hierarchy of the data structure. Note that the Artist and Tracks properties are initialized with new objects, which are then explicitly assigned from the Fox objects' child object properties. For the tracks we have to loop through all the tracks and add them to the Tracks collection explicitly.

While it's quite a bit of code, it's very simple mapping code – there's really no logic. We're just taking the result structure from FoxPro and mapping it into a .NET object structure and that's the idea when you use COM interop.

COM Interop is meant for leaving all of the business logic in the COM Component so that the component returns only the result. The service code then should only have to deal with the resulting data structure and mapping it to something that .NET can return.

Updating Data – Client and Server

We already saw how to pass data from the client proxy to the server in the original *GetAlbumsQuery()* method where we passed a filter object to the Web Service. In the next example, let's create the logic needed to update one of the artists. Let's create two new FoxPro methods on the COM server: *GetArtist()* and *SaveArtist()*. Here are both of those methods:

```
************************************************************************
*  GetArtist
************************************
FUNCTION GetArtist(lnId)

SELECT * FROM Artists WHERE Id = lnId ;
   INTO CURSOR TArtist

IF _Tally < 1
   USE IN TArtist
   RETURN null
ENDIF

SCATTER NAME loArtist MEMO
USE IN TArtist

RETURN loArtist

************************************************************************
*  SaveArtist
************************************
FUNCTION SaveArtist(loComArtist)

IF VARTYPE(loComArtist) # "O"
   RETURN -1
```

```
ENDIF

IF !USED("Artists")
   USE Artists IN 0
ENDIF
SELECT Artists

LOCATE FOR Id = loComArtist.Id

IF !FOUND()
   SCATTER NAME loArtist MEMO BLANK
   CALCULATE  MAX(id) TO lnMaxid
   loArtist.Id = lnMaxId + 1
   APPEND BLANK
ELSE
   SCATTER NAME loArtist MEMO
ENDIF

loArtist.ArtistName = loComArtist.ArtistName
loArtist.Descriptio = loComArtist.Descriptio
loArtist.Descriptio = loArtist.Descriptio +  TRANSFORM(loComArtist.Id)
loArtist.AmazonUrl = loComArtist.AmazonUrl
loArtist.ImageUrl = loComArtist.ImageUrl

GATHER NAME loArtist MEMO

RETURN loArtist.Id
```

Pretty straight forward, right? The save operation uses SCATTER NAME to load up an existing record and the replace the values from the update object that we are interested in. It then GATHERS to save the data to disk.

On the Service end we can implent these two methods also quite simply:

```
[WebMethod]
public Artist GetArtist(int artistId)
{
    dynamic fox = CreateObject();
    dynamic foxArtist = fox.GetArtist(artistId);

    // COM nulls come back as DbNull objects
    if (foxArtist is DBNull)
        return null;

    var artist = new Artist();

    artist.Id = (int)foxArtist.Id;
    artist.ArtistName = foxArtist.ArtistName;
    artist.Descriptio = foxArtist.Descriptio;
    artist.AmazonUrl = foxArtist.AmazonUrl;
    artist.ImageUrl = foxArtist.ImageUrl;

    return artist;
}

[WebMethod]
public int SaveArtist(Artist artist)
{
    dynamic fox = CreateObject();
```

```
    int result;
    result = (int) fox.SaveArtist(artist);

    return result;
}
```

These methods simply forward the FoxPro objects to the service. All the meat of the 'work' happens in the FoxPro object.

Calling the Update Methods

Let's jump back to the client now and lets see about calling these update methods from the service. In order to do this I have to add the new service we've created which is different than the original service – a different service endpoint. So I go through the import routine for the AlbumServiceCom.asmx url for importing the service.

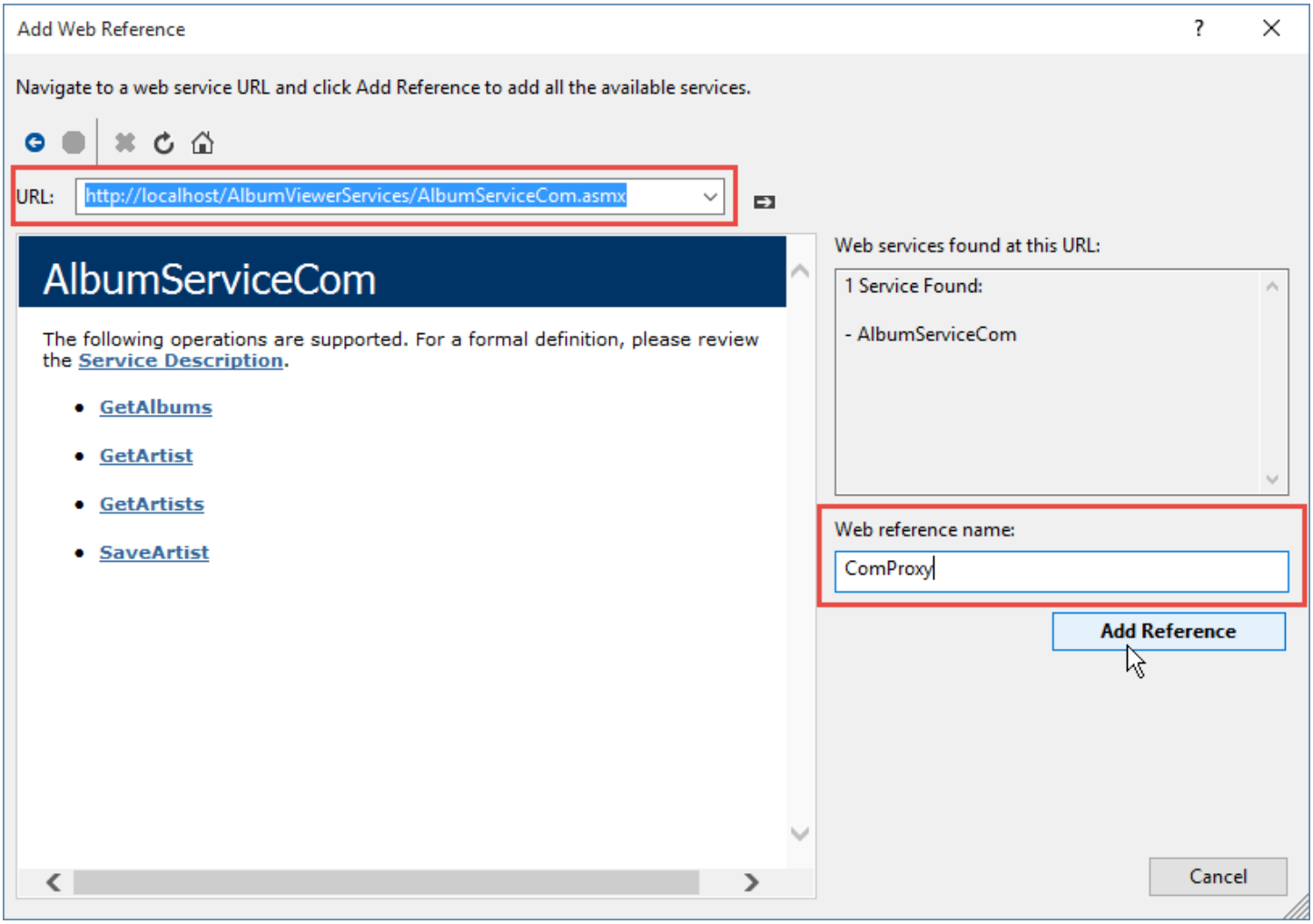


Figure 17 – Adding a second service with a new service reference/namespace. The new namespace will be completely separate from the original service imported even if type names overlap (ie. both services have an Artist class: Proxy.Artist and ComProxy.Artist which are different in the eyes of .NET even though they have the same signature)

This will give us a second proxy object in a separate AlbumServiceProxy.ComProxy namespace:

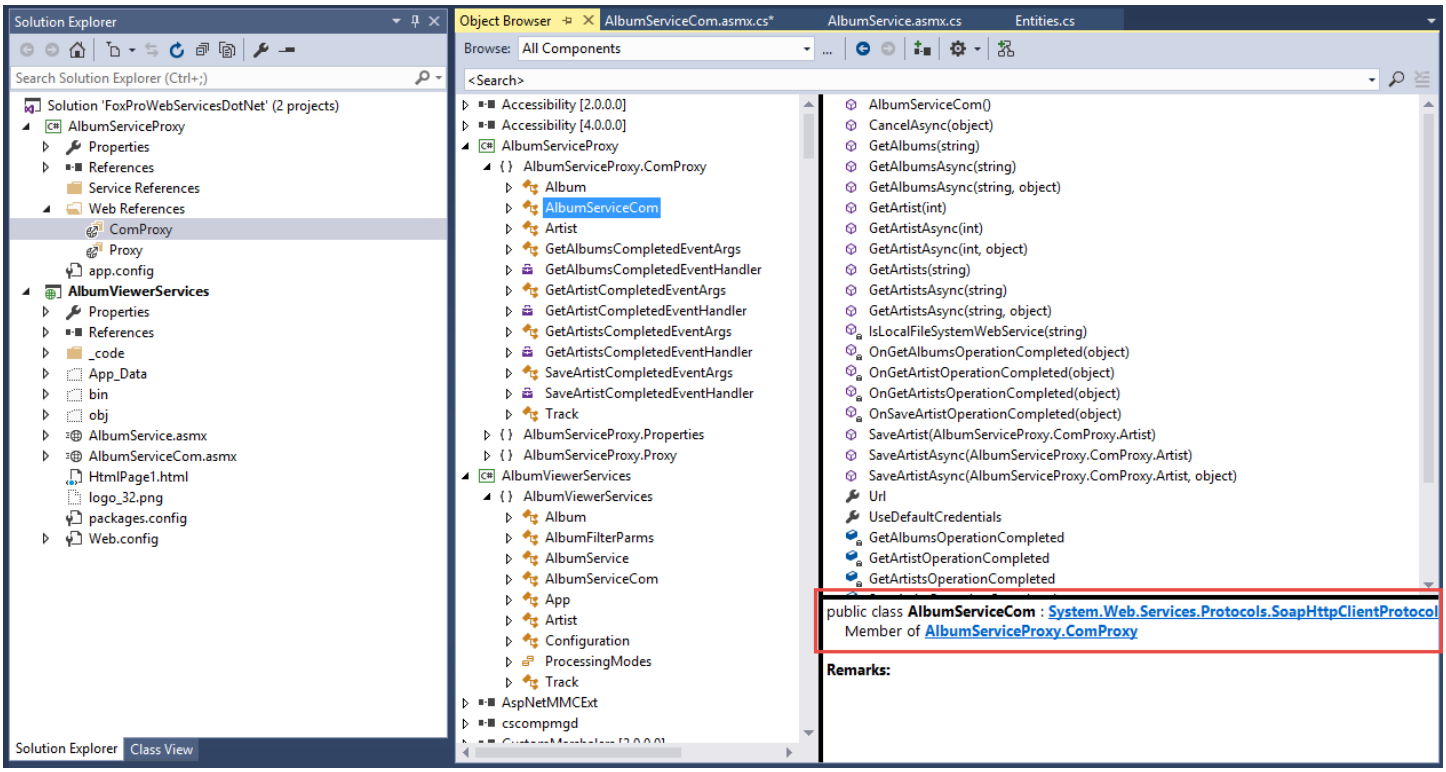


Figure 18 – See the new namespace in the Object Browser. Always verify the type and method names. Reflector provides even more info including hotlinks in properties and editors to jump to the underlying types.

So in FoxPro we can now use these new methods as follows:

```
LOCAL loService as AlbumServiceProxy.ComProxy.AlbumServiceCom
loService = loBridge.CreateInstance("AlbumServiceProxy.ComProxy.AlbumServiceCom")

*** Always check for errors!
IF ISNULL(loService)
   ? "Unable to create service: " + loService.cErrorMsg
   RETURN
ENDIF


loArtist = loService.GetArtist(2)

*** Update a value
loArtist.ArtistName = "Accept " + TRANSFORM(SECONDS())

lnId = loService.SaveArtist(loArtist)
IF lnID < 1
   ? "Failed to update..."
   RETURN
ENDIF


*** Create a new Artist
loArtist = loBridge.CreateInstance("AlbumServiceProxy.ComProxy.Artist")
loArtist.ArtistName = "New Artist " + TRANSFORM(SECONDS())
loArtist.Descriptio = "Description goes here"

lnId = loService.SaveArtist(loArtist)

? "New Id: " + TRANSFORM(lnId)
```

Notice that I'm referencing the new namespace for the second service which is different from the original service we called. Both services can exist side by side in the same assembly and they even share the same classes like Artists but in separate namespaces. This ensures that .NET gets unique objects references for each of the services you create. Note that the **`AlbumServiceProxy.ComProxy.Artist`** is not the same **`AlbumServiceProxy.Proxy.Artist`** – in the eyes of .NET these are two distinct objects even though they have exactly the same signature.

Notice also in the code above that I simply pass the loArtist instance directly – I don't need to use InvokeMethod because loArtist is a simple object. Always try using the direct methods first and if that fails, then try using the indirect methods.

So this concludes our tour of creating services and calling them with FoxPro code. There are a few additional examples not discussed here, but that are provided in the sample code that goes with this article with both the OleDb and COM code synced up.

Client Proxy Abstraction for FoxPro Service Clients

In all the client service access examples so far I've shown how the full code to create wwDotnetBridge and create a service reference for each request. That works but there's a lot of noise in that code that used for each of those requests. Ideally you'll want to abstract all of that setup code and create a single class that handles the actual service calls off a single instance that you can instantiate from your business layer or front end code to abstract away all the logic for the service access.

Likewise you should use this class to handle service errors and avoid having to handle those errors in your application level code – wrapping all service calls in exception handlers and then returning the captured error information is a good idea. I'll also show you how you can handle error messages returned from the service more effectively.

Here's what a Service Proxy class might look like:

```
*** Load dependencies
DO wwDotnetBridge
SET PROCEDURE TO AlbumServiceProxy ADDITIVE

************************************************************
DEFINE CLASS AlbumServiceProxy AS Custom
************************************************************

oBridge = null
oService = null

cErrorMsg = ""
cErrorDetailXml = ""
lError = .F.

cUrl = "http://localhost/AlbumViewerServices/AlbumService.asmx"
FUNCTION cUrl_Assign(lcValue)
this.oBridge.SetProperty(this.oService,"URL",lcValue)
ENDFUNC


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

```
*** Create wwDotnetBridge instance (latest version)
this.oBridge = GetwwDotnetBridge()

IF !this.oBridge.LoadAssembly("AlbumServiceProxy.dll")
    ERROR "Couldn't load AlbumService Assembly: " + this.oBridge.cErrorMsg
    return
ENDIF

this.oService =  this.oBridge.CreateInstance("AlbumServiceProxy.Proxy.AlbumService")

IF ISNULL(this.oService)
   ERROR "Couldn't instantiate the album service: " + this.oBridge.cErrorMsg
ENDIF

ENDFUNC
*   init


************************************************************************
*  GetArtists
************************************
FUNCTION GetArtists(lcArtistName)
LOCAL lvResult, loException

IF EMPTY(lcArtistName)
   lcArtistName = ""
ENDIF

this.lError = .f.
lvResult = null

TRY
      lvResult =  this.oBridge.InvokeMethod(this.oService,"GetArtists",lcArtistName)
CATCH TO loException
    *** Parse the exception into
    THIS.GetErrorDetail(loException)
ENDTRY

RETURN lvResult
ENDFUNC
*   GetArtists


************************************************************************
*  GetAlbums
************************************
FUNCTION GetAlbums()
LOCAL lvResult, loException

this.lError = .F.
lvResult = null

TRY
      lvResult = this.oBridge.InvokeMethod(this.oService,"GetAlbums")
CATCH TO loException
    *** Parse the exception into
    THIS.GetErrorDetail(loException)
ENDTRY

RETURN lvResult
ENDFUNC
*   GetAlbums
```

```
************************************************************************
*  GetAlbum
****************************************
FUNCTION GetAlbum(lcAlbumTitle)
LOCAL lvResult, loException

this.lError = .F.
lvResult = null

TRY
      lvResult = this.oBridge.InvokeMethod(this.oService,"GetAlbum",lcAlbumTitle)
CATCH TO loException
    *** Parse the exception into
    THIS.GetErrorDetail(loException)
ENDTRY

RETURN lvResult
ENDFUNC
*    GetAlbum

* … more methods omitted for brevity

************************************************************************
*  GetErrorDetail
****************************************
PROTECTED FUNCTION GetErrorDetail(loException,laError)
LOCAL loException, loType, lcTypeName

this.lError = .T.
this.cErrorDetailXml = ""
this.cErrorMsg = loException.Message

IF AT("OLE ",this.cErrorMsg) > 0 AND AT(":",this.cErrorMsg) > 0
   this.cErrorMsg = ALLTRIM(SUBSTR(this.cErrorMsg,AT(":",this.cErrorMsg) + 1))
ENDIF


*** Make sure we're dealing with a SOAP Exception
loException = THIS.oBridge.oDotnetBridge.LastException
IF ISNULL(loException) OR ISNULL(loException.InnerException)
   RETURN
ENDIF

IF TYPE("loException.InnerException") = "O"
      loException = loException.InnerException
ELSE
    RETURN
ENDIF

loType = this.oBridge.InvokeMethod(loException,"GetType")
lcTypeName = this.oBridge.GetProperty(loType,"Name")

IF lcTypeName != "SoapException"
  this.cErrorDetailXml = this.oBridge.GetPropertyEx(loException,"Message")
  RETURN
ENDIF

*** Grab the full XML Error Detail block
this.cErrorDetailXml = this.oBridge.GetPropertyEx(loException,"Detail.OuterXml")

ENDFUNC
* GetErrorDetail
```

```
ENDDEFINE
*EOC AlbumServiceProxy
```

The key features of this code are the Init() method that initializes the service and wwDotnetBridge and stores references to both of those objects on the class instance for reuse. This effectively isolates the server setup code to a single method that internally handles setting up the service.

Each service method is then essentially implemented with very short logic that invokes the .NET method in the generated proxy. The method call is wrapped into a TRY/CATCH handler to capture any errors and store the error message on the cErrorMsg property.

Error handling is routed to a custom method that parses the COM error from the COM Interop into an easy to read error message and also optionally captures any SOAP exception error that were sent from the service. Firing a SoapException on the service pushes the error message all the way to the client.

Notice also that this code handles the service URL with a cURL property which can be assigned to set the more transparently.

Using this proxy is now a lot simpler because all the Web service access logic is abstracted away into this separate class:

```
DO AlbumServiceProxy

loProxy = CREATEOBJECT("AlbumServiceProxy")

loArtists = loProxy.GetArtists("")
? loArtists.Count

loAlbums = loProxy.GetAlbums()
? loAlbums.Count

loAlbum = loProxy.GetAlbum("Power Age")
? loALbum.Title
? loAlbum.Artist.ArtistName
```

All of the service method calls become simple one liners and you can simply check the lError property to check for errors or otherwise just pick up the values.

I highly recommend you take this approach to create your Web Service access code – it makes your code more readable and manages setup and errors for the service all in one place. This is also what the West Wind Web Service Proxy generator automatically generates (with some other additions).

Odds and Ends

In this section I'll describe a few FAQ style items that are important to consider as you build your applications that use Web services for both the client and the server

Server Deployment

When you are building server applications that use anything FoxPro related there are a few considerations that you have to make when it comes to hosting your application. Although what I've described above uses .NET as the server platform, you are still using Visual FoxPro with the server, which means you need to have FoxPro support on the server.

FoxPro Runtimes and Libraries

Specifically this means you have to have the FoxPro runtimes and/or the FoxPro OleDb provider installed on the server. If you host the application on your company server that's probably not an issue as you can ensure that the right components are installed.

However if you plan on hosting the service with a cheap hosting provider or any multi-hosting provider you have to make sure that the right components are installed.

- FoxPro OleDb Provider
- Visual FoxPro 9 Runtimes

COM Registration Challenges

Further if you use FoxPro COM Interop you have to make sure that you can register your COM object on the server (/regsvr32) and that you can continue to update that registration anytime your COM interface for the server changes. On a shared host this is often not possible.

Even on a machine you control, updating COM objects can be challenging as you have to shut down the Application Pool (or server) in order to copy in and register a new DLL. It takes a good process to ensure you can update your COM objects on the server properly.

Data Access and Exclusive Access

If you are using FoxPro data you may also have to worry about locked data files if you need to perform EXCLUSIVE operations on the data.

The bottom line to all of this is that you probably need a dedicated server to run your FoxPro COM servers or even OleDb applications rather than using an shared server on of the low end hosting companies. An in-house server will be easiest, but any self-hosted server at an ISP or a Virtual Machine host should do as well.

Client Side Proxy Challenges

This article has talked about calling Web Services using a .NET client via COM Interop from FoxPro. As there are no official alternative choices for calling Web services for Win32 applications, this is the 'official' route to calling services. I've been working with a lot of customers in the last 10 years now using these techniques and they are reliable and work efficiently.

There are a few technical hiccups in this approach mainly related around security and access to .NET components when not running on a local machine.

Use .NET 4.x

First off you should use .NET 4.x if possible. All that I've described here works with .NET 2.0 but as of Windows 8.1 .NET 2.0 is no longer installed by default on Windows. Instead .NET 4.5 is the default install on current versions. Most other Oss that have been running for even a short while are also likely to have .NET 4.0 installed as many applications use that runtime.

If you don't support Windows XP I would actually recommend you use .NET 4.5 and later as there are major performance and security improvements. However, if you have to support Windows XP, .NET 4.0 is the latest version that supports it.

Make sure you don't have the .NET 4.0 Client Framework Installed

.NET 4.0 has an unfortunate sub-version of the .NET framework that doesn't include any of the server related features. It was meant as a smaller version of .NET but unfortunately commonly used stuff is missing in this version.

This stripped down version doesn't have support for the WSDL based SOAP client and a number of other features that are used by wwDotnetBridge internally.

Unfortunately detecting which version of .NET is actually installed is quite difficult because all 4.x versions are actually version 4.0 and only vary by their minor build numbers. This site has the best way to find what version you are running and how to get the smallest download needed to get the latest version your system supports:

- **SmallestDotNet**

Using .NET Assemblies from the Network

.NET 4.0 unlike .NET 2.0 allows EXE based applications to execute off network shares without explicitly setting execution policy. More importantly for Visual FoxPro applications (which are not .NET executables) you can also set a configuration switch in a config file to allow assemblies to be loaded from network shares.

To do this create a config file with the same name as your main EXE for the application in the same folder as the EXE. So create MyApp.exe.config like this:

```
<?xml version="1.0"?>
<configuration>
  <startup>
    <supportedruntime version="v4.0.30319"/>
    <!-- supportedRuntime version="v2.0.50727"/ -->
  </startup>
  <runtime>
    <loadFromRemoteSources enabled="true"/>
  </runtime>
</configuration>
```

You also need to do this for the Visual FoxPro IDE, so find your vfp9.exe file and create vfp9.exe.config.

Using this switch you should be able to launch your distributed .NET assemblies off a network path.

Alternatives

Are there other client options for calling Web Services from FoxPro? If you want to use a real SOAP based client – not really. If you want to use WSDL to discover service functionality and have a proxy generated the .NET tools are the only official way to call Web services and use them in FoxPro today. There are old and obsolete tools like the SOAP Toolkit but I would really discourage you from using those.

One option that you do have if you are willing to hand code XML is to just use SoapUI to test your SOAP requests and then generate SOAP envelope XML manual from your FoxPro code. If you do go down that route, please do not generate strings. Use the MSXML DOM to generate the XML to ensure that values are properly encoded. XML is not just text – it's an encoded text format and encoding is easy to break by an errand character.

SOAP 1.x Only

All of the samples I've showed in this article use the classic .NET Web Service client which supports the SOAP 1.x standard (1.1 and 1.2 are the major versions). I used this one because it's much easier to work

with via FoxPro Interop – and that's important. Using this client you can generate the classes and do nothing in .NET code and simply start firing away at the proxy client from FoxPro with Interop.

The downside of this client is that it doesn't support WS* Web services.

What about WS*

WS* SOAP Web Services are services that support extended SOAP protocols like WS Security, WS Policy, WS Federation etc. These are additional features that have been tacked onto the SOAP standard and are typically handled as part of a complex SOAP header that is added to the SOAP XML document.

To use WS* you can use the .NET WCF (Windows Communication Framework) client to call these services, but using this library is a bit more complex and requires a lot more configuration than the classic SOAP client I showed in this article. You can use the WCF client for simple SOAP calls as well as complex WS* services, but I would avoid using WCF with COM interop unless you really need the features it provides. Using WCF requires additional configuration settings in a configuration file or writing some explicit .NET code to create a properly configured service instance. WS* protocols are notoriously finicky and include tons and tons of miniscule adjustments that have to be matched exactly (and typically aren't described in the WSDL document) so just connecting and getting a message to go through with a WS* service can be a major issue.

WCF supports this functionality and like the proxy we generated above, WCF allows creation of a .NET proxy that you can access from FoxPro. However, the configuration of that proxy is much more complex and you most likely will want to delegate the creation of the actual service client to .NET code that you write, which then in turn is called from FoxPro to retrieve that reference.

It's beyond the scope of this article to discuss to show how WCF integration works, but once you have a service instance the mechanics of calling the service and retrieving results is very similar to what we've seen with the classic SOAP client using COM Interop and wwDotnetBridge.

Summary

Creating and consuming Web Services with FoxPro code or data is not a trivial task due to the fact that there are no official tools left today to support SOAP development on Win32 or COM. So you are left with the official route which is using .NET to handle SOAP based services.

In this article I've shown you how to create classic ASMX Web Services using ASP.NET using the OleDb provider to directly access FoxPro data and using FoxPro COM objects to access FoxPro business logic from the Web Service. Even if you are new to .NET using either of these approaches is pretty straight forward and in most cases should involve a minimum amount of .NET code – you are primarily mapping properties from your data results onto the service message objects that represent the service interface.

To consume the service – or **any** SOAP 1.x service really – I've shown you how you can use the .NET WSDL client to generate a .NET proxy from the Web Service and then access that .NET proxy from FoxPro using .NET Interop. Once create the proxy classes handle all the XML generation for the SOAP message and results. The process is pretty straight forward, but you have to deal with some of the type conversions necessary to access .NET code from FoxPro.

I hope this article has been useful. There's a lot of information here, but once you get the basic ideas straight, the process to create or call services using .NET is not very difficult.

Resources

- **Source Code, Slides, Notes for this Sample on BitBucket**
- **West Wind Web Service Proxy Generator**

About Rick Strahl

Rick Strahl is the Big Kahuna and janitor at West Wind Technologies located on the beautiful island of Maui, Hawaii. Between windsurf sessions and spikey haired adventures, Rick has been a software developer for over 25 years, developing business and Web applications since the very early days of the Web when you needed a hand crank (or a pair of wire splicers) to get online. Today Rick builds client centric Web applications and services for customers with HTML5, JavaScript and mobile Web technologies, using AngularJS on the front end, and the ASP.NET stack and Visual FoxPro on the back end. Rick's company West Wind Technologies also produces a number of developer related tools including West Wind WebSurge, Html Help Builder and West Wind Web Monitor. Rick is also available for consulting and mentoring services and maintains a host of open source libraries at http://github.com/RickStrahl. You can find Rick's blog at weblog.west-wind.com or contact him directly at http://west-wind.com/contact.