Version: FoxPro 7.0
Figures:
File for Subscriber Downloads:

Publishing Your First Web Service – Part II

Whil Hentzen

Last month we built our first Web Service, published it, and consumed it – all on our development machine. It wasn't a very good Web Service, but that wasn't its purpose. Its purpose was to show you the steps to take, where all the tools are, and which secret buttons to push. At the end, I promised that we'd deploy that Web Service to a live server this month, but in the interim, I've changed my mind. After going through that process a few times, the inevitable questions arise. This column answers many of those questions, and prepares us for the Real Deal – deployment on a live server – next month.

This month I'm going to go through a Q&A of the process, in order to explain what's going on under the hood, and what some of the options you have are. A lot of that information wouldn't have made sense during our first run-through, but now that you've seen the whole process, you'll be able to put some of these more advanced issues in context. Once we're done with that, we'll be more comfortable with the nuts and bolts, and then we'll be ready to deploy – yes, next month.

Proper Names

First of all, let's talk about the name of the Web Service class.

```
o=createobject("wsc.hwpclass")
? o.getnews()
```

This means that **wsc** is the project and **hwpclass** is the name of the class. And GetNews is a method of the class. The best practices suggestion for naming ProgIDs of components says that ProgIDs should follow the pattern of company.project.class. Thus, this ProgID should have been named

```
Hentzenwerke.wsc.hwpclass
```

That really doesn't tell us much, because "wsc" and "hwpclass" are dummy names for this example. Given that the purpose was to dig out news, perhaps a better name would include a project named FoxTalkDemo, and the class would be named NewsServices. Thus, the ProgID would be

```
Hentzenwerke.FoxTalkDemo.NewsServices
```

Now, how do you do this? You use the Servers page of the Project Info Dialog. See Figure 1.

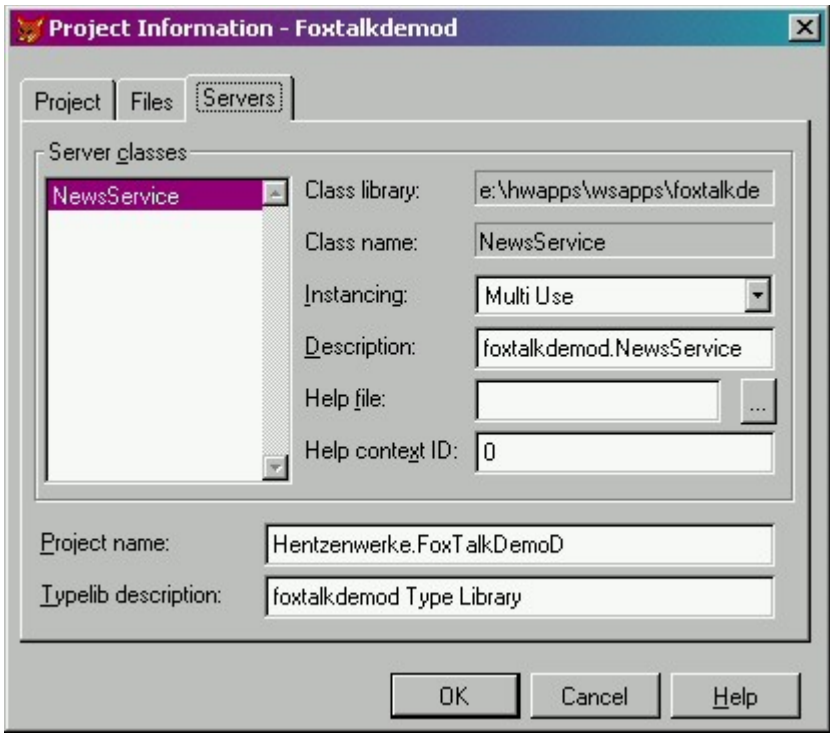

Figure 1. Naming a COM component properly.
`Figure WS19`

Registering a DLL

When you build your DLL with the Project Build dialog (Figure 3 of last month's article), you're doing more than creating a file with a .DLL extension. You're also registering the DLL in the Windows Registry. This was, when another program receives a call to create an object like

```
o=createobject("wsc.hwpclass")
```

or (in Visual Basic)

```
Dim o As New wsc.hwpclass
```

The "wsc.hwpclass" ProgID is used to look up the CLSID in the Windows Registry. The CLSID is used to look up the full path to the DLL/EXE and its type library. The type library, then, is used to determine the interface.

You can rebuild your class over and over again if you like, and the same ProgID is used – the entry in the Windows Registry is simply updated. If you change your interface, though, then you'll want to check the Regenerate Component IDs checkbox. You need a new ProgID and CLSID because you wouldn't want a user to create an instance of the new version of the component (with its new interface) using the old ProgID and CLSID. Imagine how hard **that** would be to debug!

Publishing your Web Service

Simply building your DLL, though, isn't enough to make it a Web Service. It's what logicians call a 'necessary but not sufficient' condition. A Web Service is a DLL that has been worked on some more, so to speak. We used the Visual FoxPro Web Services Publisher to publish the DLL as a Web Service. The Web Services Publisher does the following things:

- creates WSDL and WSML files
- optionally, registers with IntelliSense
- optionally, installs a project hook in the VFP project so that subsequent rebuilds of the DLL automatically call and run the Web Services Publisher.

In general, the WSDL and WSML files are similar to type libraries in VFP and VB. In other words, they describe to the outside world what methods are in the Web Service.

WSDL stands for "Web Services Description Language" – it's an XML document that describes what services are available from the Web Service on the server. It's sort of like a menu. The WSDL file also describes the format that the client must follow when requesting a service.

WSML stands for Web Services Meta Language. This is particular to SOAP 2.0, and it provides the information needed to map the operations of a WSDL-described service to specific methods in your DLL. The WSML file determines which COM object to load in order to provide the Web Service requested.

You can find out more info about these files – way more than you'll ever want to know – using the SOAP 2.0 help file, probably located in Program Files\MSSOAP\SOAP.CHM.

When you then consume a Web Service, you'll call a WSDL file, which tells you if the way you're calling the web service is legit or not. For example, if your Web Service has a method called "GetNews", the WSDL contains that info.

Here's what a part of the WSDL file looks like:

```
<portType name='hwpclassSoapPort'>
  <operation name='getnews' parameterOrder=''>
    <input message='wsdlns:hwpclass.getnews' />
    <output message='wsdlns:hwpclass.getnewsResponse' />
  </operation>
```

When you try to consume that Web Service by invoking a method called GetNewsOfTomorrow, the call will fail, because the WSDL file doesn't know anything about the GetNewsOfTomorrow method.

Select Class

When you publish your Web Service, you use the Visual FoxPro Web Services Publisher. (See Figure 8 of last month's article.) In my example, the Select Class combo was disabled. Why is it a combo, and why is it disabled?

You can have multiple OLEPUBLIC classes in a COM component, but a Web Service can only publish one of those classes. (Of course, one of those classes can expose any number of methods.) The combo is used to select which class you want to publish with this particular Web Service. If you wanted to publish more than one class, you'd need to create another Web Service.

Advanced options – saved settings

The settings in the Advanced Options dialog are saved and will be used for subsequent publications of the Web Service. They are saved in the FOXWS.DBF file, the location of which is stored in the _FOXCODE system memory variable.

Advanced options – virtual directory

In the Advanced Options dialog (see Figure 9 of last month's article), the location of your WSDL file and Listener include the name of the virtual directory that your Web Server is pointing to.

Whoa, you say. Virtual directory? Yes, I said "Virtual Directory." And if you've already done the virtual directory thing, you might still want to read. Here's the deal.

I have IIS set up so that the home directory points to E:\INETPUB\WWWROOT. But Figure 9 from last month shows that the WSDL (and, presumably, the WSML) file is in http://FOXDOTNET/webpub. I would have thought that this meant physically the WSDL and WSML files went into e:\INETPUB\WWWROOT\WEBPUB. However, when I published my Web Service, the publisher put the WSDL and WSML files in C:\INETPUB\WEBPUB. In other words, somewhere else. I was terribly confused at first.

It turns out that the first time you run the Web Services Publisher, a virtual directory is created automatically. This virtual directory is a semaphore that points somewhere else. Thus, in http://FOXDOTNET/webpub, FOXDOTNET points to the IIS root on my machine, and "webpub" is the virtual directory that points to a specific physical directory on the machine. In this particular case, "webpub" points to C:\INETPUB\WEBPUB. Note that the virtual doesn't even have to point to a location in the same vicinity as the IIS root. That's why the WSDL and WSML files ended up in C:\INETPUB\WEBPUB, not E:\INETPUB\WWWROOT\WEBPUB.

How do you set up a virtual if you haven't already done it? Or if you want to change it? First, open up IIS in the Computer Management applet in Control Panel. Right click on Default Web Site and select the Properties menu option, and click on the Home Directory tab to display the Home Directory properties as shown in Figure 2. And that's the home directory that IIS is pointing to.

In order to create a virtual directory that points elsewhere, right click on the Default Web Site node and select New | Virtual Directory to bring forward the Virtual Directory Creation Wizard welcome screen, and then click Next. In the next dialog, enter the name of the virtual directory. For example, the VFP Web Services Publishing wizard uses "WebPub" as the name of their virtual directory.

Next, point to the directory that the virtual directory will represent - in other words, the physical directory on your machine. You can either browse for the directory or just type it in the text box. Then you'll need to configure access permissions. Don't worry if you don't get them all correct – I'll show you where you can change these in a moment.

After clicking Next in the Access Permissions dialog, a "Finished!" dialog appears, and you're done. The results – a new virtual directory – will be visible in the Default Web Site tree in the Computer Management applet. Now you can go look at the virtual directory properties – and change them - by right clicking on the virtual directory node in Computer Management, and selecting the Properties menu. The properties dialog will appear, and the access permissions checkboxes are visible in the middle of the dialog.

Advanced options – ISAPI vs ASP listeners

The next option you see in the Web Services Publisher Advanced Options dialog is the choice of listener. What is a listener? A listener is a mechanism (physically, it's a DLL) that handles incoming SOAP requests on the server. You have two choices for a SOAP listener:

- An Internet Server API (ISAPI) server
- An Active Server Pages (ASP) server

When you use an ISAPI listener, then the SOAPISAP.DLL is used. When you specify an ASP listener, then the ASP page specified causes ASP to handle the request.

In the Web Services Description Language (WSDL) file, the URL identified as the server-side handler of the SOAP request determines how the SOAP request is handled on the server. For instance, the following WSDL file fragment identifies the URL that invokes the ISAPI listener:

```
<definitions>
…
  <service name='DocSample1' >
    <port name='DocSample1PortType' binding='tns:DocSample1Binding' >
      <soap:address
          location='http://localhost/DocSample1Test/DocSample1.wsdl' />
    </port>
  </service>
…
</definitions>
```

If this URL identified an ASP file instead, it would look like this

```
'http://localhost/DocSample1Test/DocSample1.asp'
```

and invoke the specified ASP script. The problem is that VFP, by default, tries to register Web services using an ISAPI listener. However, the SOAPISAP.DLL listener is not typically configured in IIS and has to be configured for ISAPI listeners to work. Go figure, right? The quick workaround is to simply configure the WS to use an ASP listener. However, Microsoft recommends ISAPI listeners for performance reasons.

Here is how to set up the ISAPI listener (for SOAPISAP.DLL), if you want to. (You can find additional details in the "ISAPI listener" topic in the SOAP.CHM file.)

First, right click on your Default Web Site in IIS, select Properties from the pop-up menu and click on the Home Directory page as shown in Figure 2:

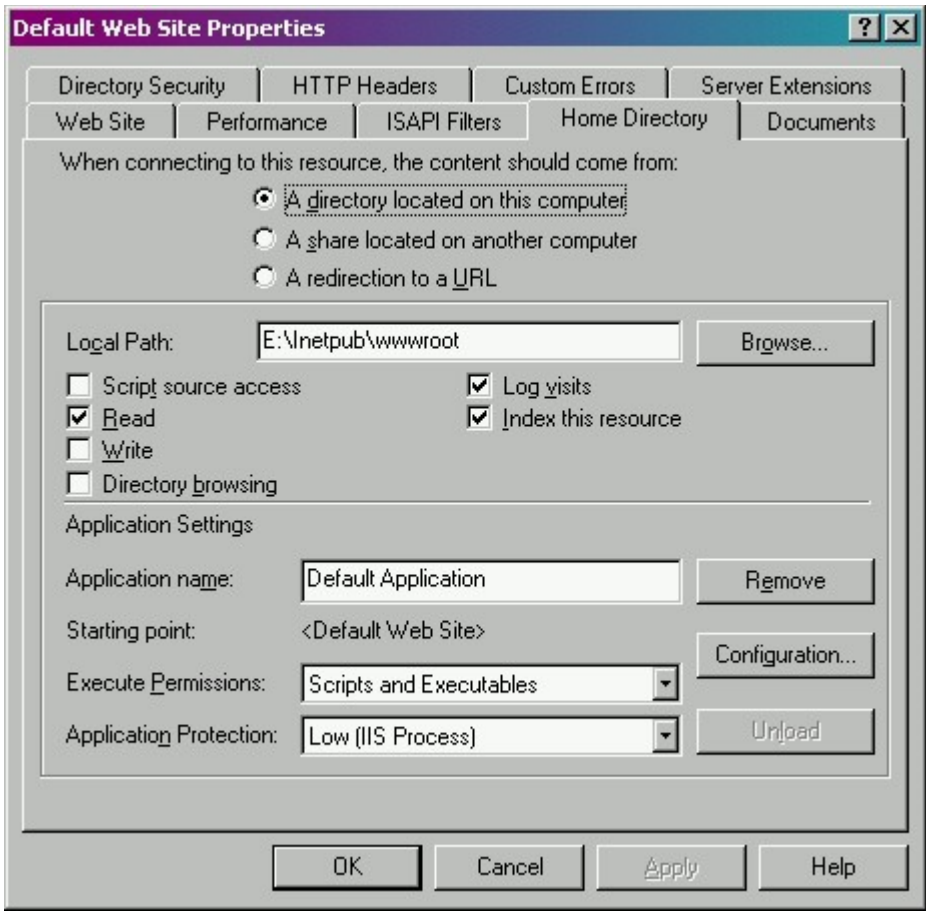

Figure 2. The Home Directory tab of the Properties dialog.

Click the Configuration button to open the Application Configuration dialog as shown in Figure 3:

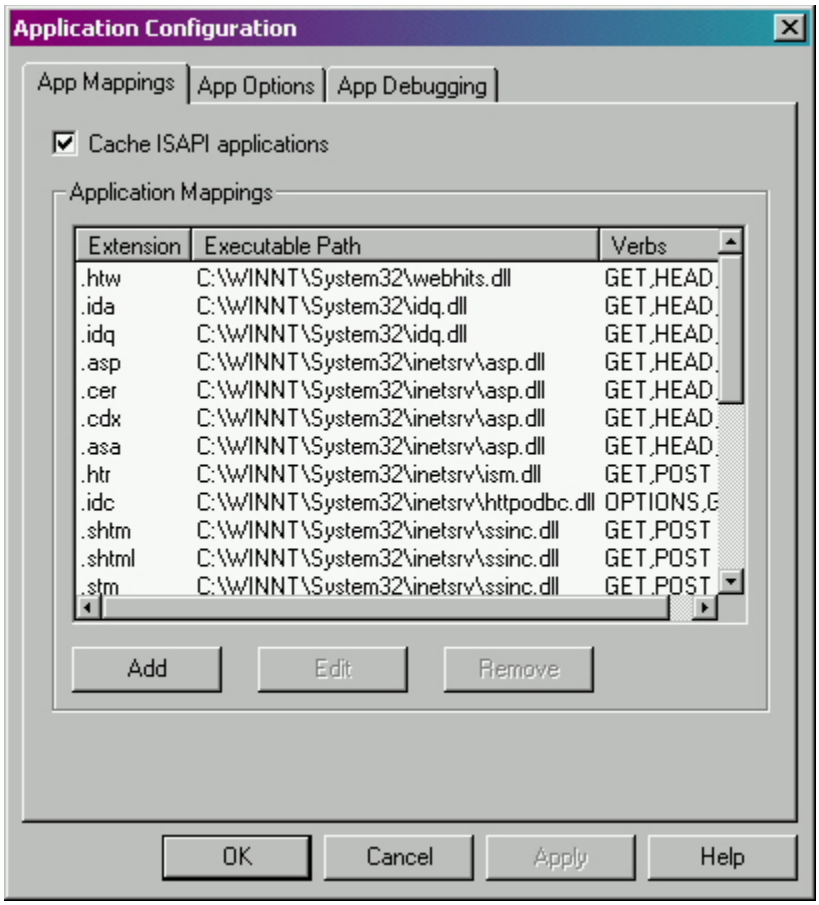

Figure 3. The Application Configuration dialog displays mappings between extensions and server DLLs.

Look for the .wsdl extension in the list. If it isn't there, click the Add button. In the Add/Edit Application Extension Mapping dialog, shown in Figure 4, click the Browse button and browse for the file "C:\Program Files\Common Files\MSSoap\Binaries\soapisap.dll" and set up everything else as shown:

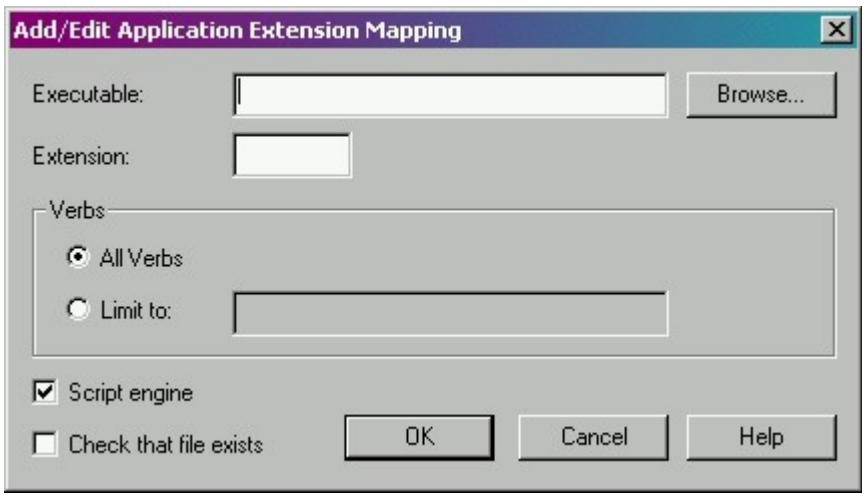

Figure 4. The Add/Edit Application Extension Mapping dialog allows you to create and edit mappings between extensions and server DLLs.

If you can't find the DLL you're looking for via the Browse button, don't fret. I've set up application extension mappings on a number of machines in both NT4 and Windows 2000 for a variety of applications, and I've never found the DLL in the File Open dialog. You can pick a similar file (often I'll use an INI file with the same name) and then edit the name of file once the Add/Edit dialog has been filled in.

Figure 5 shows you what it will look like filled in.

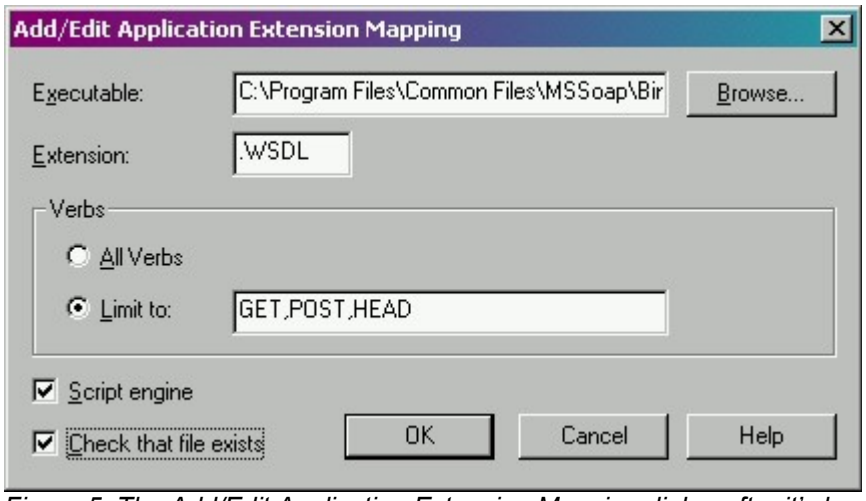

Figure 5. The Add/Edit Application Extension Mapping dialog after it's been filled in.

Gary DeWitt reports that this process works fine but he has not been able to get an ISAPI listener to work on a server running Windows XP Professional. You might need to use ASP listeners on XP boxes. However, you probably will only run into this during development, as you wouldn't want to use XP professional for an actual deployment – you'd want to use NT4 or Windows 2000 Server.

Advanced options – IntelliSense scripts

Checking the "IntelliSense scripts" check box and entering a name will create an IntelliSense entry in FOXWS.DBF. If you don't check this check box, you'll have to write out all the code by hand when you create a program using a Web Service.

You can use your own name instead of the default provided. In the example in February's article, the default was "hwpclass web service" – not very useful. If I had named the class a bit better, such as "NewsService", the IntelliSense name would have been "NewsService web service" – much better.

Advanced options – UTF-16 Unicode

English and most European languages use a single-byte character set – all of the necessary character combinations can be represented by a single byte. In other languages, such as Oriental dialects, there are more characters than can be handled by a single byte, so two bytes are necessary. Unicode is a type of double-byte character set. If you need to use a double-byte character set, you'd check this box.

Web Services Publishing Results

The next dialog shows you what happened. A COM server was created, WSDL was created, the listener was identified, and an entry in IntelliSense was created. Why do you care about IntelliSense if you're publishing a Web Service? I mean, you're creating files to deploy – why do you want IntelliSense entries on your development box? Mainly because you'll want to test your Web Service immediately after testing. You don't have – if you would be testing from somewhere else, but that's probably a fairly rare circumstance.

Web Services Registration

On the flip side, you're a client, wanting to consume a Web Service. In order to do this easily, you created an entry in the IntelliSense Manager by clicking on the Web Services button in the Types tab to open the Visual FoxPro Web Services Registration dialog. See Figure 14 in last month's article for a refresher.

This dialog allows you to name the entry being stuffed into the IntelliSense Manager as well as identify the Web Service you're interested in. In last month's article, we pointed to the Web Service we just built on the same machine; most likely when you're consuming a Web Service, you'll be pointing to another machine – somewhere else in the Internet Universe.

Specifically, this operation puts entries into FoxCode so that you can type

```
LOCAL ows as
```

pick "hwpclass web service" from the drop down list, right next to other classes like textbox and grid, and have all of the code associated with that Web Service into your program (or command window).

Why \NEWS was kludged to be found in the root (and where a COM server is run from)

When you ran the COM server from the VFP command window, like

```
o=createobject("wsc.hwpclass")
? o.getnews()
```

Visual FoxPro instantiated the class from VFP's current directory. It's not running the COM server that was registered in the Windows Registry. This is why it found data on drive E.

However, when you ran the test program that created the web service, you actually instantiated the DLL, using information in the Windows Registry. I have my test program in the root of drive E. But the COM component isn't installed or running from the root of E. Where IS it running from? From WINNT\System32! How do you know that? You could have someone tell you that, or you could figure it out with a temporary piece of code like so in your wsc.prg, just before the SELECT statement:

```
strtofile("Where is my web service DLL is running?", ;
   "myveryownwebservice.txt")
```

When you run your web service, the file "myveryownwebservice.txt" will be written in the current directory – in this case, winnt\system32.

In other words, when you run a COM component, the default directory is the directory where the Visual FoxPro runtime files are (which can be identified by looking at the HOME() function). Typically, this is the system directory – and that's NOT a good place to be sticking things…. Like your data!

So, when you install your DLL (and any supporting files), do the install in a directory of your own choosing. Then, in your code, issue the command

```
set default to (JUSTPATH(_VFP.ServerName))
```

The ideal place would be in your class's Init event, so that the rest of your code can find everything.

Thus, when the sample Web Service looked for NEWS.DBF in last month's article, it was looking in WINNT\SYSTEM32. Since, initially, NEWS.DBF was buried somewhere on drive E, well, sorry Charlie. Thus, for the sake of this demo, I explicitly pointed to the root of the current directory, and then copied the NEWS.DBF file to the root of drive C. When we go 'live' next month, we'll clean that up a bit, but for the time being, while we're trying to get our IIS virtual directory and the WSDL and all the other pieces working, the last thing we needed to worry about is some pathing issue having to do with the data itself.

Passing parameters

In the little chunk of code I wrote in last month's article, I allowed for a parameter to be passed in. While the mechanism I used works for regular old COM components, components that are published as Web Services should use the AS keyword to define parameter and return value types. If you tried to pass a parameter to the Web Service before making that change, you'd get an error message as shown in Figure 6.

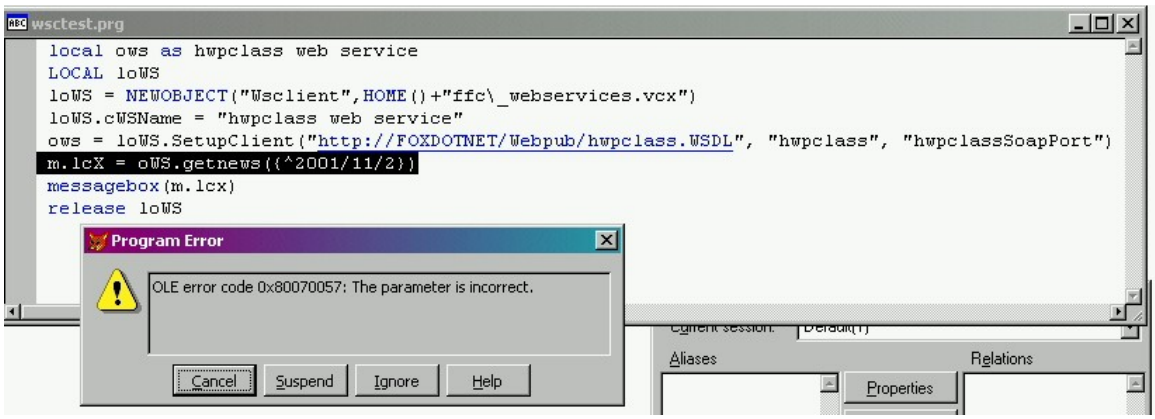

Figure 6. Passing parameters incorrectly results in an awkward error message.

Specifically, the first few lines of the class should be changed from

```
DEFINE CLASS hwpclass AS session OLEPUBLIC
Name = "hwpclass"
PROCEDURE getnews
LPARAMETERS ldDate
DO case
```

To

```
DEFINE CLASS hwpclass AS session OLEPUBLIC
Name = "hwpclass"
FUNCTION getnews (ldDate as Date) as String
DO case
```

This defines a parameter, ldDate, of type Date, and defines the return type as String as well. After that change has been performed, you need to republish the Web Service (using the Tools | Wizards | Web Services menu option). If you don't republish, you'll see the following in your WSDL file:

```
<message name='hwpclass.getnews'>
</message>
```

There is no parameter in this declaration. If the WSDL knew about the parameter, it would be listed in the XML above and would include its type, like so:

```
<message name='hwpclass.getnews'>
  <part name='ldDate' type='xsd:dateTime'/>
</message>
```

Next in the "bad" WSDL file, you'll see the following:

```
<message name='hwpclass.getnewsResponse'>
  <part name='Result' type='xsd:anyType'/>
</message>
```

The return type is "anyType", MS SOAP Toolkit lingo for "variant". This means either that the method was defined in a VCX (not true in this case – I used a PRG) or a return type wasn't specified (guilty as charged). After defining the type of the return value, the WSDL file shows the following:

```
<message name='hwpclass.getnewsResponse'>
  <part name='Result' type='xsd:string'/>
</message>
```

Much better.

Now, in WSCTEST.PRG, you'll use the line

```
m.lcX = oWS.getnews({^2001/11/2})
```

to pass a date parameter to your Web Service.

Note that you don't have to write your code in a PRG. You can use a VCX – you just need to expose it for COM or Web Services from a PRG, like so:

```
DEFINE CLASS MyCOMInterface AS Custom OLEPUBLIC
FUNCTION MyMethod (MyParm AS String) AS String
  LOCAL o AS myclass
  o = CREATEOBJECT("myclass")
  RETURN o.MyMethod(MyParm)
ENDDEFINE
```

This causes all sorts of good things to happen. First, it's a good separation of interface and implementation. (If you're still fuzzy on this concept, just think of the interface as being the properties and methods that the outside world – like users of the Web Service – can see, while the implementation is the programming magic you worked inside the component – the magic that no one else will see.)

Second, you can't define types in VCXs – only in PRGs, via the inline syntax I just showed you.

And finally, if you do this, you can define the interface, build your component, and publish the Web Service (create the WSDL, in other words) just once. You can go back and tweak the implementation without having to republish the Web Service. The COM type library will export your actual data types, not variants, and the resulting Web Service will also show the types correctly.

DLL In Use errors

You will most likely have to rebuild your DLL in Visual FoxPro over and over again. When you do so, you'll probably run into the error message "XXX.DLL is in use" and you won't be permitted to finish the build process. What's happening is that IIS has cached your module in memory in order to improve performance. Yeah, I know, it's not improving our performance as developers, is it?

OK, so the DLL is still running – how do you get it to stop? The sure fire method is to close everything down and reboot your machine, but, well, that can be time-consuming and tedious. The better way is to issue the Restart IIS command.

Open up the Computer Management applet from the Control Panel, select IIS, right-click and select the Restart IIS menu as shown in Figure 7.

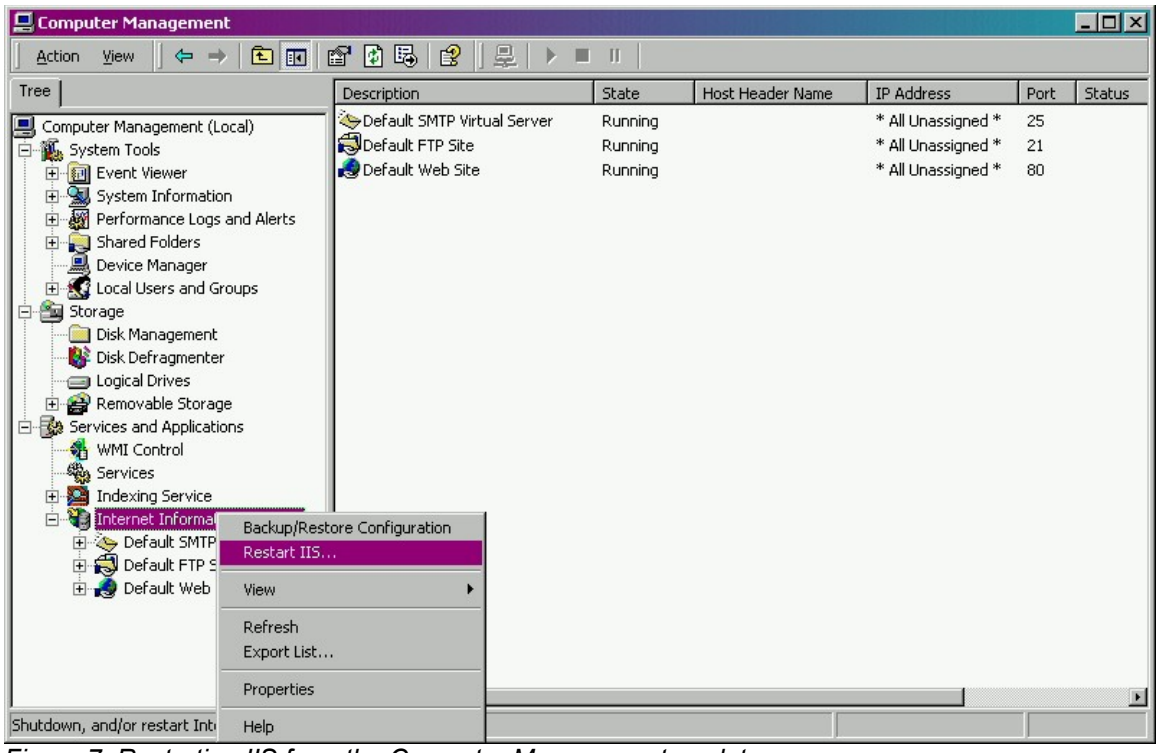

Figure 7. Restarting IIS from the Computer Management applet.

You'll be greeted with the dialog shown in Figure 8.

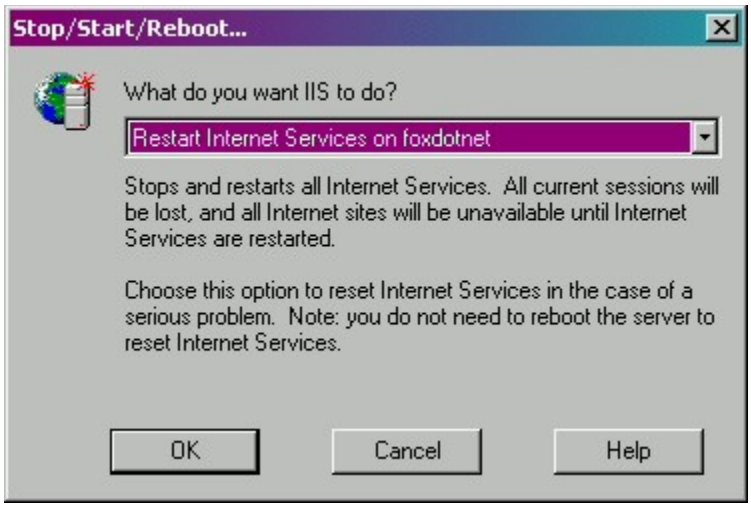

Figure 8. Selecting an option in the Restarting IIS dialog.

Select Restart Internet Services, click OK, and wait, oh, maybe a half minute or so. Eventually you'll be returned to the Computer Management screen and you'll see your services are running again. You can now rebuild your DLL.

Too much work? If you're an old codger, you can also use a DOS window to run IISRESET. It's much faster than using the GUI tool. Just issue IISRESET at the command prompt, and then hit F3 every time after that.

That's all the questions we have time for this month. Now that we've got a more thorough understanding of what's going on under the hood, we can create a more robust Web Service. Next month, that's exactly what we'll do – and we'll deploy it as well!

Whil Hentzen is editor of FoxTalk.