Muse Vs. .NET Integration - Part 3 a New Frontier

Posted At : November 9, 2010 11:47 PM | Posted By : Mark Kruger
Related Categories: Coldfusion Troubleshooting

As we continue our saga against the evil empire our intrepid Jedi, Master Muse, has a new chapter of knowledge to share. So with this post I'm adding a few things you *may* need to know to work with .NET integration on ColdFusion - in this case ColdFusion 8.01 enterprise in "multi-server" mode. Before I continue it might behoove you to read my previous post on **Muse vs. .NET integration Part I** and **Muse Vs. .NET Integration Part 2**. These 2 previous posts provide a blow-by-blow description of issues faced and resolved to get this running. My father used to say "behoove" so I pulled it out of the attic to use here. It means "It might be in your interest" but as a child I always thought it had to do with goats or fauns or something - but I digress.

Here is the latest information we have uncovered in our never ending quest to get .NET integration working smoothly on ColdFusion 8 64bit. Many thanks to fellow guru and very large brained friend Guy Rish for his tireless efforts to uncover some of these items. I'd also like to thank Dennis N for his part, a brilliant engineer in his own right who has kept at this with us till we solve it or die trying. I'd mention his last name but I don't have his permission :) Meanwhile.....

What in the Ham Sandwich is Going On Down There

Let's pause and take a moment to contemplate how this all works. On one side there is a bridge "listener" that is hooked up to the .NET Framework. This Listener takes the form of the .NET Integration service (the binary is JNBDotNetSide.exe). If you poke around in a few config files you will see that this process actually listens on port 6805. On the Coldfusion side the "createobject()" function connects to the framework through this listener and passes the class and assembly file name. It then creates a proxy jar file - much like it would do for a web service I suppose. On multi-server this jar file is typically located in \JRun4\servers***instance name****\cfusion.ear\cfusion.war\WEB-INF\cfclasses\dotNetProxy. If you call a .NET object and then take a look at this folder, you will see a fresh new jar file created. You will also see a jar file called "dotnetproxy.jar". This jar file is (presumably) the Java classes needed to communicate with our .NET listener as well as (again presumably) some reflection classes to introspect and tease out an assemblies various properties and methods.

One of the things we discovered is that it is sometimes necessary to clear this folder out - particularly when you have a new DLL you are trying to instantiate but it has the same signatures and class name(s) as the old one.

Side Trail to "Signature" Explanation

And while we are at it, that word "signature" is a good one to learn while you are absorbing .NET Integration. You see, .NET is like most other compiled languages in that each method can have more than one "signature". For example, if you wrote the following ColdFusion code in the same script:

```
<cffunction name="getUser" ...>
    <cfargument name="userID" type="numeric"../>
</cffunction>
```

```
<cffunction name="getUser" ...>
    <cfargument name="lastname" type="string"../>
</cffunction>
```

ColdFusion would immediately error out and complain that you cannot have 2 functions identically named in the same template. But in .NET (or other compiled languages) these 2 functions could happily coexist - one as "getUser(string)" and the other as "getUser(int)". In compiled languages the method name plus the return type plus the argument names and argument types make up the functions "signature". Why is this important? Because using .NET integration you will likely have to match the signature of the method you are calling. Let's say the .NET method needs the following:

```
addUser(
 string,
 string,
 string,
 boolean,
 int,
 string,
 date,
 string,
 double)
```

If you pass the wrong number of arguments, get the arguments out of order, or miss an argument altogether you might expect an error like "Hey - you forgot argument X".... but that's not what you will get. Instead you will likely get an error like "No method found with that signature". For example, let's say you tried this:

```
<cfscript>
    AddUser('mel','Torme','A','true',85,68154,'6/6/1942',2000.00);
</cfscript>
```

It looks ok but there *might* be 2 things possibly wrong. Can you see them? First, the Boolean in position 4 is actually being passed as if it were a string with quotes around it. Secondly, the item in position 6 (supposed to be a string) which looks like a zip code is being passed as a numeric value (no quotes). In a ColdFusion function call this would likely pass muster. CF would convert the number to a string and the string to a Boolean. Or, perhaps, CF would throw an error like "argument blah must be a Boolean". But in .NET you will get the "no method found with that signature" error instead - which might leave you scratching your head.

In other words, .NET will not call the method and simply "pass" whatever you throw at it. Instead the method you call has to match the arguments precisely by type. And .NET is not going to give you extra clues either. You will have to eyeball your signature Vs. the one expected by .NET. Here's a tip. If you can get the assembly instantiated dump it out:

```
<!--- instantiate --->
    <cfset obj = createObject(".NET","someclass","#pathtocache#/blah.dll")/>
     <!--- dump the object --->
     <cfdump var="#obj#"/>
```

You will get a list of distinct method/signatures. Make sure your call to the .NET method matches a signature in the dump. If you get a mismatch error try "JavaCasting"

all the non-strings into the appropriate java type. It takes some trial and error but it's doable.

Back to Clearing the Cache

I'll explain why it's necessary to clear the cache in a moment, but let's talk about the "how". If you try to delete these little jar files you will get a complaint that they are locked - and indeed they are, by the .NET integration service. To delete them you will need to *stop* the integration service, delete the jar files (remember to leave the dotnetproxy.jar file in place) and restart the service.

But Why? Why are you Taking our Jar Files Santy Claus?

Sit down here on my knee Cindy Lu and I'll tell ya. You see it all started with the need to support 2 different dlls with the same class names. In our case we had site A - that used the "current" implementation of our .NET dll. We also had site B - which desired to use the latest and greatest release of our dll. Each of these sites used the same CFC to make the .net calls. Being geniuses we naturally had a fool-proof plan. We put together something like this:

```
<cfscript>
    IF(site IS 'A')
        createobject('.NET',"mainclass","c:\prod.dll");
    else if(Site IS 'B')
        createobject(".NET","mainclass","c:\beta.dll");
</cfscript>
```

Simple right? If we are on Site A we instantiate "prod.dll" and if we are on site B we instantiate "beta.dll". Each of these dlls has the same methods attached to a class called "mainclass". Some of you might be ahead of me already. Remember that our .NET assembly is compiled into that jar file in the /dotNetProxy folder. And jar files work as a collection of classes bundled together. You put a jar file in the class path and the JVM "knows" about the classes contained within it. If you follow the logic here you will see that one class is going to be instantiated and a jar file built with a class at *.cfclasses.dotnetproxy.mainclass. But when site B comes along it's going to build it's own jar file with, guess what, the same class name - *.cfclasses.dotnetproxy.mainclass. So one of these classes is going to overwrite the other in the class path and be the only one available to use.

Conclusions

One very obvious conclusion is that you cannot use 2 versions of the same class - the same limitation you have with Java. If 2 classes have exactly the same path and dotted notation, one is going to "win out" and the other will not be usable. And that, dear readers, is why we had to "clear the cache". In point of fact we kept overwriting one class with another and back again till we figured out what we should have seen from the beginning - there was only room for one class of that name in dotnetproxy. Why didn't we see it sooner? That's an instance question. The cache and class path are specific to the instance. In our tests we had logically used an instance that was NOT the production instance. In other words we had an instance at \JRun4\servers\ProdInstance\.... and another at \JRun4\servers\TestInstance.... - 2 JVMs, 2 Class paths, 2 cfclasses/dotnetproxy folders, and no conflict. We only saw the issue when we tried to run the 2 classes simultaneously on the same instance.

Go Easy

Now perhaps some smug reader might say "well sure... everyone knows that". I admit in hindsight it seems obvious (sort of), but it was hard to get our heads around the fact that we could instantiate 2 different file names and not have separate classes. Once we figured that out (and slapped our foreheads repeatedly - Guy has a permanent hand print there) it was much easier to conceptualize and draw conclusions. But as you know, the Muse enjoys helpful and insightful comments, so fire away.