# Muse Vs. .NET Integration - Part 1

Posted At : October 19, 2010 12:23 AM | Posted By : Mark Kruger
Related Categories: Coldfusion Troubleshooting

When Adobe (then Macromedia) came out with Coldfusion 8 one of the oft touted features was the .NET integration service. The idea was to provide the same easy-to-use accessibility that create object *used* to give to COM (although COM itself was unstable) and *still* gives to Java and web services. Just like ColdFusion gives you handy access to the universe of Java, the .NET integration service was designed to give you equally handy access to the world of .NET assemblies and managed code.

In practice however, I found that few developers chose to use it as a solution. Why? I think one reason is likely the emergence of Web Services and SOAP as a practical intermediate middle layer between various technologies - and especially between .NET and Java. When the integration service worked it was an effective painless solution. When it did *not* work however, it proved difficult to troubleshoot and configure. The fact that it was not a commonly adopted solution meant that fewer developers where asking questions about it or choosing it as solution to .NET-to-Java integration problems.

The Muse and his merry men (and women) were no exception in this regard. As an active ColdFusion shop doing over 1000 hours of ColdFusion consulting each month, we have worked with .NET integration only a handful of times in the years since its release. In each of those cases it was very simple implementation. The .NET service was usually chosen because of the use of client certificates or some other special requirement that included a bit of managed code that was not easy to duplicate in CF. In most cases, however, the web service implementation meant more "pure" CF code and better compatibility. For that reason the Muse never really delved into how this service was set up or how to troubleshoot or configure it. Until recently that is....

## The Setup

A recent customer of ours had moved from Coldfusion 8 32bit to Coldfusion 8 64bit. As a part of this migration this customer had elected to install ColdFusion 8 in "multi-server" mode to give it the added flexibility of being able to spool up and cluster ColdFusion instances as needed. As CF Guru Mike Brunt is fond of saying, a good many things are really not even possible till you settle on Multi-Server (or J2EE) as your preferred configuration. In any case, this dual change from standard install to multi-server and from 32bit to 64bit caused some major headaches for them in the realm of .NET integration. While I didn't know it at the time there were actually several problems and each of them had to be solved to affect a repair.

For security reasons this customer had created a .NET assembly to serve as an interface to a set of remote web services. The Web Service in question used a number of different internal service URLs and name spaces. A pure ColdFusion or even Java implementation would have been fairly tedious to create and require much trial and error. Since they already had a C# sample, they wrapped up the functions they needed in an assembly and utilized the .NET integration service to access the data. The code was very straightforward. It wasn't full of complex properties or nested object calls. Rather, it was a collection of methods each of which took a list of primitive arguments (strings and numbers). So it wasn't so much of an "object" as an interface or library. The code was similar to this:

```
<Cfscript>
    //create .net objec
    obj = createObject(".NET","someclass","#pathtocache#/blah.dll");
    // access function
    user = obj.getUser(firstname,lastname);
</CFSCRIPT>
```

## The Problem

Upon moving the code to the new platform this process simply ceased working. Examining the OS showed that the same version of .NET was on the server and sample .NET code (without CF) could be run from the server. The issue was simply that .NET integration was not accessible. No matter what we tried (at this point) we could not get the object to instantiate.

Consulting with some .NET folks we decided that at least one issue was that the .NET assembly was compiled as 32bit and it needed to be recompiled as 64bit to be accessible from the 64bit JVM and OS. The .NET service runs (by default) on a TCP port and it is capable of running using shared memory (with the JVM) or SOAP/HTTP. I was hesitant to hang my hat on the idea that the service couldn't handle 32bit managed code. It *did* run on the server after all - if communication was purely TCP port based, why would it matter. After all, Verity runs as a 32bit service with socket based communication - right? Still, the .NET guys were pretty insistent so we went down the path of recompiling to .NET 64bit. This proved to be doable with a few studio changes and some new references. The new .DLL tested out all right, but we *still could not access it from ColdFusion*.

## Interim Proxy Magic

While .NET folks wrangled with recompiling and the host tried to install and reinstall the .NET integration service, the server process continued to fail *on production*. Obviously something had to be done and fast. We needed a stop gap measure while we assessed the situation. Rolling back to 32bit was not an option at this point - that ship had sailed. So what could we do? I would add that it was now 1:00 in the morning. My good friend in the hosting business had been trying to solve the issue since 9:00 pm. The customer needed resolution before they opened for business the following morning. But desperate times call for desperate measures. Our clever solution was as follows.

- **32 Bit VM** - We spooled up a 32bit VM of ColdFusion 8. We installed our .NET assembly and verified that it worked with the ColdFusion .NET integration services.
- **Proxy CFC** - We built a proxy CFC on the 32bit JVM with remotely accessible methods. For each method in the .NET assembly we created an identically configured method in our CFC. For example, if the .NET object had a method with the signature "getUser(string firstname, string lastname)", we would also have a "getUser( )" function with 2 string cfarguments - firstname, lastname. These identical methods would collect the arguments, pass them correctly to the .NET assembly using the right syntax and then return the results of the Assembly to the calling process. In other words, they were identical 'proxies' for the .NET object.

- **Re-point CreateObject Calls** - On the 64bit server we had reengineered each of the .NET calls to make a remote call to our newly created proxy service. So code that looked like this:

```
<Cfscript>
    //create .net objec
    obj = createObject(".NET","someclass","#pathtocache#/blah.dll");
    // access function
    user = obj.getUser(firstname,lastname);
</CFSCRIPT>
```

...now looks like this:

```
<Cfscript>
    //create .net objec
    obj = createObject("webservice","*http path to 32 bit WSDL");
    // access function
    user = obj.getUser(firstname,lastname);
</CFSCRIPT>
```

With very minimal code changes (a pretty easy search and replace in a couple CFCs) we were able to move the .NET processing to our separate proxy server. The plan was, once we get our 64bit .NET integration code working again, we can easily roll this code back to .NET calls with minimal impact.

## The Nuances

Although this plan worked well, there were some difficulties we had to overcome. The new proxy process required several steps and if you have any experience with SOAP, Axis and .NET you will likely recognize a few of them. The problem is multiple layers of data type translation. Data has be marshaled through these various layers and translated into types that the layer can understand. Think about the steps for a moment.

1. **Call to local CFC on 64bit** - ColdFusion accepts the call to its own local function with a list of ColdFusion arguments specified by type in the CFARGUMENT tag.
2. **Call to Webservice on 32bit** - ColdFusion translates its native datatypes ("String" or "Numeric") into SOAP based datatypes with the help of the Axis library (compliments of the Apache project folks I believe).
3. **Call to Proxy CFC** - The Proxy CFC translates this SOAP back into ColdFusion native types, then passes it to the .NET assembly.
4. **Call .NET Assembly** - In order to work with the data the .NET assembly creates .NET native types from the Java types (or these types are translated through the JNI layer).
5. **Call .NET external web service** - Now the data is translated into SOAP using native .NET libraries and sent to the external web service.

**Back to .NET** - External .NET gets the arguments, translates them into native .NET, does it's thing and creates response data in native .NET.

- **Response Data** - the response data goes back in reverse order through the stack till it arrives at the original calling CFC on the 64bit server.
  If you are paying attention there are no fewer than 11 translations occurring here - CF-SOAP-CF-.NET-SOAP-.NET-SOAP-.NET-CF-SOAP-CF. It's really kind of mind boggling that it works at all.

  As you might imagine most of the problems that arise are problems with data or types being "lost in translation". We found that at the proxy level we needed to use JavaCast( ) on each of the arguments to insure that the native Java type "under the hood" reflected a type that .NET would translate correctly before casting it into SOAP for the external .NET service. Even though our "cfargument" had a specified type we still needed to cast all types that were not "string" by default. So we had a little section in our proxy code whose job was to recast the data as needed. Like so:

```
<cffunction name="getUser"...>
        <cfargument name="firstname" type="string">
        <cfargument name="lastname" type="string">
        <cfargument name="age" type="numeric">
        <!--- massage data --->
        <cfset age = Javacast("int",arguments.age)/>
        ....
    </cffunction>
```

Secondly, the data coming back from .NET was an array. On the local CF Server the array de-serialized into an array type and was able to be accessed via array functions and looping. But this same array directly serialized into SOAP and sent unfiltered back to the 64bit server would cause a parsing error on the original side. So before we sent the array back to 64bit server we had to add a little "rebuilder" loop to make sure it was an actual native ColdFusion array. So code that looked like this :

```
<!--- Get data from .NET object --->
    <cfreturn obj.getUser(firstname,lastname)/>
```

...now looks like this.

```
<!--- Set up a return array --->
    <cfset arrRet = arraynew(1)/>
    <!--- Get the .NET array --->
    <cfset response = obj.getUser(firstname,lastname)/>
    <!--- repopulate the CF array for return --->
    <cfloop array="#response#" index="i">
        <cfset arrayAppend(arrRet,response[i])/>
    </cfloop>
    <!--- Return this cleansed CF array --->
    <cfreturn arrRet/>
```

This allows us to strip out anything funky from the .NET version of the array and insure that our CF-to-CF remote call matches exactly.

**Permanent Fix**

Once we had our proxy cfc in place our production system was back on line and functional. This gave us the time to figure out the issues with .NET integration in the 64bit .NET ColdFusion 8 world. You will be happy to know that we absolutely *did find a solution* and it involves several specific changes and a 64bit recompile of the assembly (as our .NET developer suggested). So, Muse readers, keep your appetite whetted and I will fill you in on the fix tomorrow.