Working With IBM's MQSeries and Coldfusion MX

Posted At : April 11, 2006 11:59 AM | Posted By : Mark Kruger
Related Categories: Coldfusion MX 7, Coldfusion Upgrading

This post may be one that very few of my readers will care about. But if you are the 1 reader in 1000 that needs to know how to connect to MQSeries version 6 using coldfusion then this post may prove a life saver. You can benefit from the 50 hours of my life I spent figuring this out that I will never get back. Here's the scoop. We have a client who needs to upgrade a Coldfusion installation running on CF 5. The current installation uses COM. Under a load it becomes unresponsive.

*NOTE: There is an **update to this post** that was entered on 4/24.*

Moving to CFMX *would* seem like a viable alternative except for the fact that the site is so heavily dependent on COM. If you don't know already, COM and CFMX are uneasy bedfellows. COM is a "Windows World" product (and one that is being rapidly replaced by .NET). In CF 5 access to COM was done directly through the Windows OS using native calls, but in CFMX access to a COM object is done through JNI (The "Java Native Interface").

Think of a translator at the UN. Suppose John Bolton says "We must have more intimate international cooperation on intellectual property." Some guy in a booth quickly translates that for the North Korean delegation. Now, even if he's a good translator he might end up with something like "On brain ownership we should have more group hugs among nations." If the North Korean ambassador chooses to ask a question it might come back as "please explain how to embrace a national brain" or something equally unintelligible. Even when everything is functioning as designed things get lost in translation.

In the same way JNI takes objects and method calls from the JVM and says "hmmm... how do I package this into something COM will understand." If it is successful it must then take the result of the method call (or whatever) and package it *back into Java* so the JVM can understand. As you might imagine this process of exchanging data across completely different technologies or environments (technically called "marshalling" I believe) does not always work well. Differences in the environments mean that not every intended method or action of the COM is always possible in the JVM. It also adds a level of overhead - a middle layer - that affects the speed of the process.

So one of our first tasks was to remove this ubiquitous call to the COM object and replace it with appropriate Java code. I dug into MQSeries and found a jar file (com.ibm.mq.jar) that had all the classes I needed to duplicate the COM methodology. Here's the process I was duplicating:

1. Open the Queue Manager
2. Open a workflow *put* queue
3. Set some options
4. Send a message to the *put* queue
5. Open a *get* queue
6. Set some options
7. Using the message ID get the appropriate message back from the *get* queue.
8. Close the *put* and the *get* queue
9. Disconnect from the queue manager

Using the COM this process was a matter of creating a single COM instance and using methods attached to that 1 instance to handle the whole process.

Java and Objects

In JAVA however the process is different. It is rarely a matter of setting a few attributes. Instead you usually need to instantiate multiple objects and pass them to one another to make everything work. Hopefully you can learn from my trial and error. The process I followed was to review the documentation at publib.boulder.ibm.com for the package (jar file) "com.ibm.mq" that I was intending to use. I took my events (listed above) 1 by one and used the documentation along with createObject() and CFDUMP to step through this process - examining the return values along the way. Here are my notes.

Open the Queue Manager

The first task was to open the queue manager. The queue manager will have a "dotted notation" string for a name (like a domain name). You may need to open MQ Explorer to figure it out. The string represents the listener or "channels" on the MQSeries installation you are trying to access. Let's call it "MQDEV.QUEUE.MAN" for our sample code.

```
<Cfscript>
...
// Create a "manager" object
manager   =
createObject("JAVA","com.ibm.mq.MQQueueManager").init("MQDEV.QUEUE.MAN");
....
</CFSCRIPT>
```

Notice the "init()" part of the createobject() call. That's pretty typical. It's the "constructor" - the thing that builds the object in memory for you. Sometimes an ojbect can have more than 1 constructor and the difference is in the arguments that are passed to it. This combination of method and argument is called a "signature". For example "init(string queuemanager)" is a different signature from "init(string queuemanager, int version)". In our case "init()" takes only 1 argument - the name of the queue manager.

Open the *Put* Queue

There are different kinds of Queues to work with but the one we want is a "Workflow" queue that allows us to "put" or "send" a message. To open the queue you need to know it's name. Use MQ Explorer to see the names of the queues that are available to use. We will use "WORKFLOW.PUT.MYQUEUE" in our sample:

```
<cfscript>
   // Create a "PUT" queue object
put = manager.accessQueue("WORKFLOW.PUT.MYQUEUE",4);
</cfscript>
```

Notice that this "signature" is a little different. The put queue is retrieved as a queue object by calling a method on the manager object. The signature calls for 2 arguments - a queue name and an "option" to be passed as a part of the method call.

Create an *Input* Message Object

Ah... this is new. In COM this object was a reference we copied directly from our instance. In Java we have to create a new one and set it's properties.

```
<cfscript>
   // Create a "PUT" message object
PMsg = createObject("JAVA","com.ibm.mq.MQMessage").init();
</cfscript>
```

Note the signature on this "init()". It doesn't take *any arguments*. It creates an empty message for us to work with. We have to *write something* to it to make it useful. To do this we will use the writeUTF() function.

```
<cfscript>

PMsg.characterSet = 12;
   // message
str   =   'Buenos Dias Terra Firma';
   // write the message to the message object
PMsg.writeUTF(str);
</cfscript>
```

Notice the additional step of setting the "characterset" attribute. This item adjusts the "codepage" that MQSeries uses to unpack your message. It must know how your bytes are ordered and what they mean. If you do not set this attribute I believe the default codepage will be used. The "writeUTF" function ensures us that the message object now contains our message. It's time to pass it to the queue. Darn it, we need still *another* object to do that - an "options" object.

The "Put Options" Object

The call to the "put" queue requires that we have a *put* options object. This object can contain other attributes - although we are not using it in this case. In this case we are going to create a "blank" or "empty" options object and pass it to the queue along with our message. The main purpose of the *put* options object is to set the message up as a *put* rather than some other kind of event.

```
<cfscript>
   // Create a "put options" object
POPt   =   createObject("JAVA","com.ibm.mq.MQPutMessageOptions").init();
</cfscript>
```

As you might expect, when we go to *get* from the queue we will need a *get* options object.

Send the Message to the *Put Queue*

Finally, we are ready to send. We have a message and an options object. We just need the "put" command against the queue.

```
<Cfscript>
// put the message in the queue
put.put(PMsg, POPt);
</CFSCRIPT>
<Cfdump var="#put#">
```

The CFDUMP of the *put* object will reveal a property called "messageID". It's a cryptic signature that allows this message to be uniquely identified in the queue. Keep an eye on it because we will need it for our next operation - getting something back from the *get* queue.

The Get Queue

At least part of this operation will look familiar. We have to create the queue object, a blank message object and an options object for example.

```
<cfscript>
   // get message object
GMsg = createObject("JAVA","com.ibm.mq.MQMessage").init();
   // get message options
GOpt   =   createObject("JAVA","com.ibm.mq.MQGetMessageOptions").init();
   // create a "GET" object
get    = manager.accessQueue("WORKFLOW.GET.MYQUEUE",1);
</cfscript>
```

This is where things get a little trickier. In the case of the *get* queue we are actually going to use the options object. We are going to set the "waitInterval" and the "options" flag.

```
<Cfscript>
   // Set the wait Interval (time to wait in milliseconds)
GOpt.waitInterval = 10000;
   // Set the "options" to 1
GOpt.options = 1;
</CFSCRIPT>
```

Now our problem is synchronization. Remember a "queue" is a lineup. There are messages "waiting in the queue" ready to be picked up. Obviously we *don't want* messages that resulted from other processes. We *only* want our own message. To get our message we have to set the messageID on the "get message" object to the same messageID as the one we sent. That messageid is contained in our "put" object. So we do the following:

```
<cfscript>
   // Set message id - this tells the queue to "get"    //the first message in queue that
matches.
GMsg.messageId = PMsg.messageId;
   // set character set for consistency
GMsg.characterset = 12;
</cfscript>
```

One big gotcha here is that you cannot do something like this:

```
<cfscript>
   //THIS WON'T WORK    // reference to messageID
myId = PMsg.messageId;
   //set the messageID
Gmsg.messageId = myId;
</cfscript>
```

Why won't this work? Because when setting the messageId to a CF variable it get's passed *by value* and CF converts it from a Java "byte array" to a "String" (in CF). The property "messageId" in the *get* queue must be of the type "byte array". Since there is no "javaCast" function for "byte Array" there is no conversion back into the appropriate type. If you leave it as a property attached to the "put" object it will be passed to the GMsg object *by reference* and not converted - it will stay as type "byte array". When working with Java in CF this is one of the many pitfalls.

Go Get the Message From the Get

This should look familiar. Only instead of a "put" we are calling "get".

```
<cfscript>
   //get the message from the queue

get.get(GMsg, GOpt);
</cfscript>
```

Now our *get* object contains a message (we hope). The only thing left is to read it.

```
<cfscript>
   // get message length
MsgLen = GMsg.getMessageLength();
   // read in the message
Msg = GMsg.readString(MsgLen);
</cfscript>
```

MQSeries "MQMessage" object comes with a lot of "read" functions. Most of them are type specific (readInt(), readFloat()) depending on the type of message you expect to receive. In our case (and probably most cases) the "readString()" function gives us the entirety of the message and allows us to unpack it however we need to do it. You will also notice that we needed to get the message length before we tried to read the message. The readString function allows you to read "part" of the message based on length and character position. Since we wanted the whole message we passed in the total length.

The Final Code

Here's the final result:

```
<cfscript>
   // Create a "manager" object
manager   =   createObject("JAVA","com.ibm.mq.MQQueueManager").init("MQDEV.QUEUE.MAN");

   // PUT Operations    // Create a "PUT" queue object
put = manager.accessQueue("WORKFLOW.PUT.MYQUEUE",4);
   // Create a "PUT" message object
PMsg = createObject("JAVA","com.ibm.mq.MQMessage").init();
   // set codepage
PMsg.characterSet = 12;
   // message
str   =   'Buenos Dias Terra Firma';
   // write the message to the message object
PMsg.writeUTF(str);
   // Create a "put options" object
   POPt   =   createObject("JAVA","com.ibm.mq.MQPutMessageOptions").init();
   // put the message in the queue
put.put(PMsg, POPt);

   //GET Operation    // get message object
GMsg = createObject("JAVA","com.ibm.mq.MQMessage").init();
   // get message options
GOpt   =   createObject("JAVA","com.ibm.mq.MQGetMessageOptions").init();
   // create a "GET" object
get   = manager.accessQueue("WORKFLOW.GET.MYQUEUE",1);
   // Set the wait Interval (time to wait in milliseconds)
GOpt.waitInterval = 10000;
   // Set the "options" to 1 (not sure why - from legacy code...
GOpt.options = 1;
   //the first message in queue that matches.
GMsg.messageId = PMsg.messageId;
```

```
   // set character set for consistency
GMsg.characterset = 12;
   //get the message from the queue

get.get(GMsg, GOpt)
   // get message length
MsgLen = GMsg.getMessageLength();
   // read in the message
Msg = GMsg.readString(MsgLen);

   // CLOSE OBJECTS    // close the put queue
put.close();
   //close the get queue
get.close();
   // close the manager object
manager.close();
</cfscript>
```

Now before you write in and tell me all about CF Gateways and JMS please note that I was tasked with converting a legacy system where MQSeries is the cornerstone. I didn't have the options of saying to them "let's just chuck MQSeries altogether shall we". I'm sure there are better ways to do this. I'm equally confident that a CFX tag would be a better "black box" approach. Still, it *is possible* to work with Java and Coldfusion and they do cooperate pretty well. This solution will cause the least amount of pain and suffering, and because the COM was in a custom tag wrapper it will be a seamless transition.

Also note that the methodology used above is certainly not the best use of message queues. A message queue should be a "fire and forget" technology - where work is handed off to another process and then picked up later. The way the process is used above is more closely akin to database calls, where the "get" result from the "put" is important to the current request. Not only does this go against the whole idea of a message queue (distributed workflow), it produces a fundamental bottleneck (the queue itself) that is hard to overcome and devilishly difficult to troubleshoot.

If you are working on a Coldfusion MQSeries project you have my sympathies. I hope this post will help. If you have other ways of doing this or would like to suggest a different approach I welcome comments (as always).