Java in ColdFusion - a Festivus for the Restofus

Posted At : February 23, 2009 1:52 PM | Posted By : Mark Kruger
Related Categories: ColdFusion

Using ColdFusion means leveraging the power of Java. ColdFusion encapsulates a large percentage of the Java web universe and makes it available to you through native tags, but a vast array of additional functionality is also available through the use of Java Libraries. The truth is, however, that working with Java in ColdFusion can be frustrating for the non Java programmer. Sure there are some easy things you can utilize like getClass() and toString(), but what if you have some Java sample code and you just want to pull it into your Coldfusion page? How do you do that? Here's a short example I created in response to a recent **CF Talk** post. The setup is that Authorize.net sent a memo to everyone telling them that SSL 2.0 would no longer be supported as of late March. There was general agreement that CF 8 could handle SSL 3.0 but what about 6 and 7?

It turned out that both 6 and 7 are compatible as well (as of 1.4.02), but along the way I got the idea to grab the Java code from Authorize.net and give it a whirl in Coldfusion as a sort of test run at bypassing cfhttp. I've done this many times with other products as well (custom libraries, IBM's MQ series etc). So I wrote this post to show the process I went through to unpack Java Code into ColdFusion.

The Code

Here's a portion of the sample Java I found from authorize.net (note - I'm putting it in cfscript blocks so Ray's blog software will parse it and display it for me).

```
<cfscript>
import javax.net.ssl.*;
import java.io.*;
import java.net.*;
import java.util.*;
import java.security.*;

public static void main(String[] Args){

try{
// standard variables for basic Java AIM test
// use your own values where appropriate

StringBuffer sb = new StringBuffer();

// mandatory name/value pairs for all AIM CC transactions
// as well as some "good to have" values
sb.append("x_login=yourloginid&"); // replace with your own
sb.append("x_tran_key=eoXaDm2LUnz2OiyQ&"); // replace with your own
sb.append("x_version=3.1&");
sb.append("x_test_request=TRUE&"); // for testing
sb.append("x_method=CC&");
sb.append("x_type=AUTH_CAPTURE&");
sb.append("x_amount=1.00&");
sb.append("x_delim_data=TRUE&");
sb.append("x_delim_char=|&");
sb.append("x_relay_response=FALSE&");

// CC information
sb.append("x_card_num=4007000000027&");
sb.append("x_exp_date=0509&");

// not required...but my test account is set up to require it
sb.append("x_description=Java Transaction&");


// open secure connection
URL url = new URL(
"https://test.authorize.net/gateway/transact.dll");
// Uncomment the line ABOVE for test accounts or BELOW for live merchant accounts
// https://secure.authorize.net/gateway/transact.dll
```

```
/* NOTE: If you want to use SSL-specific features,change to:
HttpsURLConnection connection = (HttpsURLConnection) url.openConnection();
*/

URLConnection connection = url.openConnection();
connection.setDoOutput(true);
connection.setUseCaches(false);

// not necessarily required but fixes a bug with some servers
connection.setRequestProperty("Content-Type","application/x-www-form-urlencoded");

// POST the data in the string buffer
DataOutputStream out = new DataOutputStream( connection.getOutputStream() );
out.write(sb.toString().getBytes());
out.flush();
out.close();

// process and read the gateway response
BufferedReader in = new BufferedReader(new InputStreamReader(connection.getInputStream()));
String line;
line = in.readLine();
in.close();     // no more data
System.err.println(line);



// ONLY FOR THOSE WHO WANT TO CAPTURE GATEWAY RESPONSE INFORMATION
// make the reply readable (be sure to use the x_delim_char for the split operation)
Vector ccrep = split("|", line);

System.out.print("Response Code: ");
System.out.println(ccrep.elementAt(0));
System.out.print("Human Readable Response Code: ");
System.out.println(ccrep.elementAt(3));
System.out.print("Approval Code: ");
System.out.println(ccrep.elementAt(4));
System.out.print("Trans ID: ");
System.out.println(ccrep.elementAt(6));
System.out.print("MD5 Hash Server: ");
System.out.println(ccrep.elementAt(37));


}catch(Exception e){
e.printStackTrace();
}
}
</cfscript>
```

Now someone is bound to point out that perhaps the simplest solution is to take the code and wrap it in a Java class containing precisely the elements I need to use in my Coldfusion code. Indeed that *is* an excellent solution. But there are times when this is not practical. Moreover, there are many CF programmers who would love to use Java but do not have the requisite Java skills - so for them this is a practical exercise.

Object Heaven

Java is all about objects. Everything in java implements the object class. In CF, all scopes are a structure right? As in variables.x and request.y and form.a etc. This is because under the hood ColdFusion *is Java*. In fact every object you create in ColdFusion has some sort of Java object as underpinning. For example try this:

```
<cfset testvar = "African Or European Swallow?"/>

<cfdump var="#testvar.getBytes()#"/>
```

You will see output that is tagged as "binary" and looks like this:

```
6510211410599971103279114326911711411111210111111032831199710810811111963
```

What did you just do? You converted an object of type "string" to an object of type "binary". This sort of functionality is used to pass data around, convert objects or hack into the Vatican. Anyway, the first step in our code is to figure out what objects we are going to need. Because we have to figure out a way to create each of the required java objects to get our code running. So take a moment and look at the Java code above before we get going.

The URL Object

Hmmmm.... here's where we run into our first snag. The code we are trying to duplicate is the following:

```
URL url = new URL(
    "https://test.authorize.net/gateway/transact.dll");
```

Well that is just maddeningly unhelpful to us poor CF programmers. I know that "new" and the "URL url" are telling Java to create an object of type URL. I also know that at least one constructor for "URL" takes a string as an argument. But how to do this in Coldfusion? Working with Java in coldfusion requires that I set up a class using "createObject". The path part of create object should look like "java.something.something" right? Where do I get the front part of this "URL" object? And there's another problem. The word "URL" is its own scope in ColdFusion so I cannot use it.

The answer is in the "import" statements at the top. Remember those?

```
import javax.net.ssl.*;

import java.io.*;

import java.net.*;

import java.util.*;

import java.security.*;
```

These import statements are what makes it possible to ignore the path inside the Java code. When the java programmer says "URL url = new URL" Java sifts through the classes that have been imported looking for the right one. So our first task is to find one of these paths that fits and is a likely candidate to contain the constructor for the "URL" object. I'm betting on "java.net". So my first try is going to be to tack on "URL" to that path as in:

```
<cfscript>
    objUrl = createobject("java","java.net.URL");
</cfscript>
```

When I ran this code it did NOT throw an error. FYI there are 2 types of errors you will likely get at this point, either a "class not found" error, when you've guessed wrong about the path, or a "constructor not found" error, where the object cannot be created without additional information (constructor stuff). And that, dear readers, brings us to our second hurdle. We don't just need a Java "URL" object. Somehow we need to load it up with a URL?

Unfortunately this is where coldfusion can fall flat. You will have to know about the constructors for the class and do a lot of trial and error. There is, however, one tip that sometimes works. Many common Java classes (especially the native libraries that ship with Java like java.net) have an "init()" function as a constructor. So one of the things to try is to tack on "init" after the create call, add in the URL string and see what you get. Like so:

```
<cfscript>
objUrl = createobject("java","java.net.URL").init("https://test.authorize.net/gateway/transact.dll");
</cfscript>
```

What do you know! This actually works and I'm left with a fully instantiated object that has methods and properties and everything. What's the next step?

The Connection Object

Well, looking at the Java code the next step is to create an object of type URLConnection (the code looks like *"URLConnection connection = url.openConnection()"*). Rats... does that mean we have to figure out a new constructor? Actually no. The method url.openConnection() will return an object of that type, so all we really need to do is assign a new variable to that returned object as in:

```
<cfscript>
conn = objUrl.openConnection();
</cfscript>
```

Did it work? You can find out by accessing methods that belong to the connection objet - setDoOutput(), setUseCaches() etc. // connection conn = objUrl.openConnection(); //set some props conn.setDoOutput(true); conn.setUseCaches(false); conn.setRequestProperty("content-Type","application/x-www-form-urlencoded");

The DataOutputStream Object

Looking back at our original Java the next step is going to be to create a DataOutputStream object (that's the Java code that shows *"DataOutputStream out = new DataOutputStream(connection.getOutputStream())"*). No problem. Using what we found out in the previous step this should be easy. We simply do:

```
<Cfscript>
//set ouptput
    dtOut = conn.getOutputStream();
    dtOut.write(Javacast("String",str).toString().getBytes());
    dtOut.flush();
    dtOut.Close();
</CFSCRIPT>
```

I've added the additional method calls against the dataOutputStream object. Note, this object is manipulating our original object by reference. This is the object that "sends" the information to our URL. It's really wierd I know, because it makes no reference to HTTP or to the initial URL object - which is where I would think a send operation would take place. Also note, I'm "JavaCasting" a string and using the "getBytes()" method to send it downstream as byte array, then I'm flushing and closing the connection object. This is where it pays to close your eyes and just try to duplicate the Java without trying to read too much into it (which would require some kind of input reader anyway).

The InputStreamReader Object

We are almost home. We have used an output stream to send our data to the server, now all that is left is to get the results. We need 2 more objects - an "InputStreamReader" object which takes the object returned from connection.getInputStream() as its init() argument, and a BufferedReader object which takes our input stream object as its init() argument. The Buffered Reader object has a readline() method we can use to show us what has been returned from our call to the URL.

```
<cfscript>
inS = createobject("java","java.io.InputStreamReader").init(conn.getInputStream());
inVar = createObject("java","java.io.BufferedReader").init(inS);
retVar = inVar.readLine();
</cfscript>
```

Now dump out the retVar and you should see whatever authorize.net has sent back to you.

Conclusion

I know it seems like a lot of rigmarole, but a half an hours worth of trial and error produced Coldfusion code that correctly called the transact gateway for authorize.net using only Java libraries. Of course I would hasten to add that just a little bit of Java and the ability to compile your own classes will extend this capability exponentially.

Finally, I would note that this post is intended as an expose on a *ColdFusion* programmer unpacking Java classes without using Java or any great knowledge of Java. So please don't post comments about all the wonderful ways I got it wrong or could have done A or B. This post is really about the process of looking at a Java class and unpacking it into CF (if possible). It is not about the strengths or weaknesses of the java.net class or any other java library. As always, your tips and insights are welcome if they help expand knowledge.

Combined Code

If you want to run this code yourself you will need to get a test login and transaction ID.

```
<cfparam name="form.x_login" default="*add your login*"/>

<cfparam name="form.x_tran_key" default="*add your transasctionID*"/>

<cfparam name="form.x_version" default="3.1"/>
<cfparam name="form.x_test_request" default="TRUE"/>
<cfparam name="form.x_method" default="CC"/>
<cfparam name="form.x_type" default="AUTH_CAPTURE"/>
<cfparam name="form.x_amount" default="1.00"/>
<cfparam name="form.x_delim_data" default="TRUE"/>
<cfparam name="form.x_delim_char" default="|"/>
<cfparam name="form.x_relay_response" default="FALSE"/>
<cfparam name="form.x_card_num" default="4007000000027"/>
<cfparam name="form.x_exp_date" default="0509"/>
<cfparam name="form.x_description" default="TEST JAVA TRANS"/>


<Cfset str = ''/>
    <cfloop collection="#form#" item="f">
    <Cfset str = str & f & '=' & form[f] & '&'/>
</cfloop>


<Cfoutput>#str#</CFOUTPUT>

<cfscript>
    //get ulr obj
    objUrl = createobject("java","java.net.URL").init("https://test.authorize.net/gateway/transact.dll");
    // connection
    conn = objUrl.openConnection();
    //set some props
    conn.setDoOutput(true);
    conn.setUseCaches(false);
    conn.setRequestProperty("content-Type","application/x-www-form-urlencoded");
    //set ouptput
    dtOut = conn.getOutputStream();
    dtOut.write(Javacast("String",str).toString().getBytes());
    dtOut.flush();
    dtOut.Close();
    // set input
    inS = createobject("java","java.io.InputStreamReader").init(conn.getInputStream());
    inVar = createObject("java","java.io.BufferedReader").init(inS);
    retVar = inVar.readLine();
    </cfscript>
<br>


<h4>Authorize.net Sent back</h4>
<p><cfoutput>#retVar#</cfoutput></p>
<br>
<br>
```

```
<h4>Fun with dumping</h4>
<Cfdump var="#objUrl.getClass()#"/>
<Cfdump var="#objUrl.getClass().getName()#"/>
<cfdump var="#conn#"/>
```