WebSphere Application Server

WebSphere Liberty Batch Using IDCAMS

Version Date: May 20, 2016
See "Document change history" on page 14 for a description of the changes in this version of the document

IBM Software Group
Application and Integration Middleware Software

Written by:
David Follis
IBM Poughkeepsie
845-435-5462
`follis@us.ibm.com`
Don Bagwell
IBM Advanced Technical Sales
301-240-3016
`dbagwell@us.ibm.com`

Many thanks go to the WebSphere Batch team for getting all this work done.

Table of Contents

Introduction

While talking to customers about migrating their traditional z/OS batch applications to WebSphere Liberty Batch, one of the questions that comes up is how to handle IDCAMS. This is a very commonly used utility that has a wealth of capabilities.

If you rewrite a traditional batch application that is, perhaps, a mix of custom Cobol (or maybe even Assembler) code and calls to utility programs like IDCAMS, how do you move that to Java?

In this paper we will take a two simple IDCAMS examples and move them to a JSR-352 compliant Java batch job running inside WebSphere Liberty.

Let me emphasize, this is just a simple example to show you how it can be done. It is not intended to provide a production-ready method to invoke IDCAMS in all its glory from in Liberty Batch. Along the way we will point out things you might want to consider in expanding on this example to use in your environment.

A Simple IDCAMS JCL Job

We are going to do something very simple. We're going to use IDCAMS to copy the contents of a sequential dataset into another, pre-allocated, sequential dataset. Here's the JCL:

```
//STEP1  EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*

//SYSIN    DD *
   REPRO INDATASET(USER1.DFSORT.TEST1) -
       OUTDATASET(USER1.DFSORT.TEST2)
```

That's it. Let's quickly look it over. The first line is just the step in the job that runs the IDCAMS program. I've assumed this is probably part of a larger job and this one step copies a dataset that has some contents into another dataset that got pre-allocated somehow.

The datasets are both nice simple fixed block files 80 bytes wide. I dumped some random text into the TEST1 file just to have something to copy.

The next few lines in our JCL send any output to the job output. Getting to the messages produced by IDCAMS is something we'll have to handle when we run under Liberty.

And finally we have the control statements that tell IDCAMS what to do. All we want it to do is copy the contents of the input dataset to the output dataset. The datasets have 'DFSORT' in the name because I used the same sequential datasets I used in the DFSORT example that does this same copy operation.

There is a veritable wonderland of other control statements you can specify. We'll get to some fancier stuff in a little bit.

A 'Matching' Simple Liberty Batch JSL Job

To submit a JSR-352 batch job we need some JSL (Job Specification Language) that defines the job. As with the JCL, we're just going to show one step since our assumption is that this is really just part of a larger job. Here's the JSL:

```
<step id="STEP1">
      <batchlet ref="com.ibm.batch.utilities.IDCAMS">
            <properties >
                  <property name="SYSIN" value='{"statements":[
                  "REPRO INDATASET(USER1.DFSORT.TEST1) -",
                  " OUTDATASET(USER1.DFSORT.TEST2)"
                  ]}'/>
            </properties>
      </batchlet>
</step>
```

As with the JCL we have a step called STEP1. Instead of running the IDCAMS program directly we are running a Java program called IDCAMS that is in the com.ibm.batch.utilities package. That class implements the batch interface *batchlet*.

The rest of the JSL is a property provided to our batchlet. As we'll see, the JZOS interface to IDCAMS allows you to add lines of command input to build up the whole set of instructions to IDCAMS. We want to have those lines right here in our JSL, just like we had them as instream data in our JCL.

Doing that requires a small amount of cleverness. We're going to define a property for the batchlet and the value of that property will be passed to the batchlet when it gets control. Our property is called SYSIN. It could be anything we want, but that maps nicely to the JCL DD name.

The value of our property is a JSON string. We're going to need to pass in multiple lines of text and a JSON string containing an array of strings is the easiest and most extensible way to do that.

Our JSON string starts with a curly brace, as all JSON strings do. We then have a name/value pair. The name is *statements*. The name is in quotes to delimit it. The name is separated from the value by a colon.

The value of *statements* is an array of strings and an array is indicated by a square bracket. Each string in the array needs to be enclosed in quotations and the elements are separated by a comma. If we just had one line of input we could just enclose it in quotes and be done. But since we have two lines we have two strings, in quotations, separated by a comma. Remember that the IDCAMS continuation character (the dash) has to be at the end of the first string.

So really what we've done is taken normal IDCAMS input, put quotations around each line and tacked a comma onto the end of all the lines but the last one. We have to break things up into multiple lines because IDCAMS has a line length limit.

The Code Behind the JSL

We aren't going to go through all the details of the Java code in IDCAMS. The complete listing is in an appendix. In this section I just wanted to take a look at the most important parts. You will need some other things to get it to compile. And since we aren't looking at the whole listing we won't necessarily go in the same order things appear in the whole listing.

Property Injection

The first interesting bit in the code is how that property from the JSL turns up in the Java code. You do it like this:

```
    @Inject
    @BatchProperty(name = "SYSIN")
    String sysin;
```

That incantation sets up a variable called *sysin*. This variable will be automatically given the value specified in the JSL. What if the property isn't set in the JSL? Then no value is assigned to the variable. Some error handling code for that case would be nice to have.

The JZOS interface to IDCAMS

JZOS has very nicely provided a Java class that wraps the actual IDCAMS utility and gives us a lot of help using it. You can find documentation about the class here:

http://www.ibm.com/support/knowledgecenter/api/content/SSYKE2_8.0.0/com.ibm.java.zsecurity.api.80.doc/com.ibm.jzos/index.html?locale=en

The JZOS team has also graciously written some of little sample programs that use the IDCAMS API. I 'borrowed' from their sample for this little program. You can find the samples inside a zip located here:

ftp://public.dhe.ibm.com/s390/java/jzos/jzos_samples_sdk_700.zip

Using the AccessMethodServices API

The API around the IDCAMS utility is called AccessMethodServices. The interface is pretty simple. You first new up an instance of the object. Then you can call the *addInputLine* method to add a line of command input to the utility. And then just call the *execute* method to invoke IDCAMS.

```
    AccessMethodServices ams = new AccessMethodServices();
    ams.addInputLine(statement);
    int rc = ams.execute();
```

The two pieces we need to handle are taking that JSON string we provided in the JSL and turning it into statements we can pass to *addInputLine*, and then fetching any messages issued by IDCAMS to put into the Liberty Batch job output.

Parsing the SYSIN data

To parse the JSON string that got passed to us from the JSL property we'll use the EE7 javax.json package. For this to work we'll need to be sure that the appropriate feature is enabled in the target Liberty server. Do that by specifying this feature:

```
<feature>jsonp-1.0</feature>
```

On to the code...

```
    JsonReader reader = Json.createReader(new StringReader(sysin));
    JsonObject sysin = reader.readObject();
    reader.close();

    JsonArray statements = sysin.getJsonArray("statements");

    for (JsonValue jsonValue : statements) {
        String statement = jsonValue.toString();
        // Peel off leading and trailing double-quote
        String quotelessStatement = statement.substring(1, statement.length()-1);
       ams.addInputLine(quotelessStatement);
    }
```

This is actually pretty simple. The first three lines take our input property (placed in the 'sysin' String) and read it into a JsonObject. We know that the contents of the JSON string should just be one name, 'statements', and that it contains an array of strings. We call *getJsonArray* on our object and get back an array of values.

Once we have the array we can just iterate over them converting each array value to a string. There's just one catch. The values come to us still wrapped in those quotation marks we had to use to separate the strings in the JSL. A quick use of *substring* and that's taken care of. We just add each now-quoteless-string into the input for AMS.

Logging Messages

When IDCAMS is run from JCL the SYSOUT DD contains messages telling you about what the utility did. Running under the JZOS AccessMethodServices API the same messages get generated, but we need a little extra code to make sure those messages go to the joblog for the Liberty Batch job we're running. This bit of code extracts the messages from the AccessMethodServices object and logs them.

```
         String out = ams.getOutputLines();
         log.log(Level.INFO, out);
```

Setting the Exit Status

When a batchlet step completes in a JSR-352 batch job the return value is used as the exit status for the step. This is a string value. To set our exit status we will turn our return code (rc) value into a string and return it:

```
String exitStatus = new Integer(rc).toString();
return exitStatus;
```

That value could be used as part of the next step determination. We didn't show it in our JSL above, but as part of our step definition we should indicate what step comes next. That decision can be based on the exit status set for the step by this code. Presumably if the copy was successful you would want to continue to a step that processes the copy. If the copy failed then perhaps the job should be marked failed or other recovery processing might occur in some other step in the job.

If this was the only step in the job then we might want to set the exit status for the entire job based on this value. Doing that requires a couple of extra things. First of all, we need to inject one more thing. In this case we aren't injecting the value of a JSL property. Here we need to inject a reference to an object owned by the Liberty Batch container. To set the exit status for the job we need a reference to the Job Context. This code handles the injection:

```
@Inject JobContext jobCtx;
```

And then we just need this one line at the end of processing (actually right between the two lines we showed just above) to set the exit status value to our stringified return code.

```
jobCtx.setExitStatus(exitStatus);
```

A Look at the Output

When IDCAMS runs from JCL it writes messages into the SYSOUT of the job. When our JSL runs we saw the code that gets the IDCAMS messages from the AccessMethodServices object and logs them into the job log.

I ran both jobs and captured the output from both. I pruned off the JCL specific messages and the Liberty Batch messages to get just the actual messages issued by IDCAMS. This is the output from one of the runs. Can you tell which one?

```
REPRO INDATASET(USER1.DFSORT.TEST1) -
 OUTDATASET(USER1.DFSORT.TEST2)

IDC0005I NUMBER OF RECORDS PROCESSED WAS 16

IDC0001I FUNCTION COMPLETED, HIGHEST CONDITION CODE WAS 0



IDC0002I IDCAMS PROCESSING COMPLETE. MAXIMUM CONDITION CODE WAS 0
```

This is the output from the Liberty Batch JSL job run. But there is no way to tell that because it is exactly the same as the output when run from JCL.

Enhancements You Could Make

As we said at the beginning of this paper, our intent here was to just show that IDCAMS could be invoked in a Java Liberty Batch environment and do the same things you are used to it doing in a traditional JCL batch environment. We didn't intend to provide a production-ready utility you could use out-of-the-box to work with IDCAMS. What would you need to do to our example to get there?

First of all, we've hard-coded the input to IDCAMS in the JSL. That's pretty normal from a JCL perspective, but remember that you can pass in property values using job parameters. With a little extra work in the JSL we could set the batchlet property to a job property value and get the job property from job parameters supplied when the job is submitted. Then whatever is submitting the JSL would supply the IDCAMS input. That would make this into a really generic IDCAMS job.

Finally, the Java code could do more error checking. It makes no attempt to verify that a valid value (or any value at all) was provided for the property. We don't check that we got a value. We don't handle JSON parsing errors. What if it isn't an array of strings? What if those strings are longer than is allowed as input to IDCAMS?

Some toughening up in this area would be useful. Remember that the exit status can be a string so you can use it to describe the problem, not just set an ambiguous return code number for 'something bad happened'.

Make it Do Something Else!

Ok. Just for fun let's give our Java Batch IDCAMS utility something more involved than just a simple REPRO. Another normal thing to do with IDCAMS is define a VSAM dataset. I lifted this input to IDCAMS from a sample job used to set up a VSAM dataset for a CICS sample program.

Here's the original JCL:

```
//STEP1  EXEC PGM=IDCAMS
//SYSIN    DD *
   DELETE USER1.EXMPLAPP.EXMPCAT PURGE CLUSTER
   SET MAXCC=0
   DEFINE CLUSTER (NAME(USER1.EXMPLAPP.EXMPCAT)-
          TRK(1 1) -
          KEYS(4 0) -
          RECORDSIZE(80,80) -
          SHAREOPTIONS(2 3) -
          INDEXED -
          ) -
          DATA (NAME(USER1.EXMPLAPP.EXMPCAT.DATA) -
          ) -
          INDEX (NAME(USER1.EXMPLAPP.EXMPCAT.INDEX) -
          )
//*
//SYSPRINT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//AMSDUMP  DD SYSOUT=*
```

And here's our 'matching' JSL step:

```
   <step id="STEP1">
         <batchlet ref="com.ibm.batch.utilities.IDCAMS">
               <properties >
                     <property name="SYSIN" value='{"statements":[
                     "DELETE USER1.EXMPLAPP.EXMPCAT PURGE CLUSTER",
                     "SET MAXCC=0",
                     "DEFINE CLUSTER (NAME(USER1.EXMPLAPP.EXMPCAT)-",
                     "  TRK(1 1) -",
                     "  KEYS(4 0) -",
                     "  RECORDSIZE(80,80) -",
                     "  SHAREOPTIONS(2 3) -",
                     "  INDEXED -",
                     "  ) -",
                     "  DATA (NAME(USER1.EXMPLAPP.EXMPCAT.DATA) -",
                     "  ) - ",
                     "  INDEX (NAME(USER1.EXMPLAPP.EXMPCAT.INDEX) -",
                     ")"
                     ]}'/>
               </properties>
         </batchlet>
   </step>
```

Same stuff, just with those quotes and commas separating the lines. And here's the output (from the JSL run in Liberty Batch).

```
DELETE USER1.EXMPLAPP.EXMPCAT PURGE CLUSTER

IDC3012I ENTRY USER1.EXMPLAPP.EXMPCAT NOT FOUND
IDC3009I ** VSAM CATALOG RETURN CODE IS 8 - REASON CODE IS IGG0CLA3-42
IDC0551I ** ENTRY USER1.EXMPLAPP.EXMPCAT NOT DELETED

IDC0001I FUNCTION COMPLETED, HIGHEST CONDITION CODE WAS 8


SET MAXCC=0

DEFINE CLUSTER (NAME(USER1.EXMPLAPP.EXMPCAT)-
  TRK(1 1) -
  KEYS(4 0) -
  RECORDSIZE(80,80) -
  SHAREOPTIONS(2 3) -
  INDEXED -
  ) -
  DATA (NAME(USER1.EXMPLAPP.EXMPCAT.DATA) -
  ) -
  INDEX (NAME(USER1.EXMPLAPP.EXMPCAT.INDEX) -
)

IDC0508I DATA ALLOCATION STATUS FOR VOLUME WSLSMW IS 0

IDC0509I INDEX ALLOCATION STATUS FOR VOLUME WSLSMW IS 0
IDC0181I STORAGECLASS USED IS OPENMVS

IDC0001I FUNCTION COMPLETED, HIGHEST CONDITION CODE WAS 0



IDC0002I IDCAMS PROCESSING COMPLETE. MAXIMUM CONDITION CODE WAS 0
```

Just like you get in regular JCL using IDCAMS.

Appendix: Full Source Listing

```
package com.ibm.batch.utilities;

import java.io.StringReader;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.batch.api.BatchProperty;
import javax.batch.api.Batchlet;
import javax.batch.runtime.context.JobContext;
import javax.inject.Inject;
import javax.json.Json;
import javax.json.JsonArray;
import javax.json.JsonObject;
import javax.json.JsonReader;
import javax.json.JsonValue;

import com.ibm.jzos.AccessMethodServices;

public class IDCAMS implements Batchlet {


   private static final Logger log = Logger.getLogger( IDCAMS.class.getName() );

   @Inject JobContext jobCtx;


    /**
     * Default constructor.
     */
    public IDCAMS() {
        // TODO Auto-generated constructor stub
    }

   /**
     * @see Batchlet#stop()
     */
    public void stop() {
        // TODO Auto-generated method stub
    }

   /**
     * @see Batchlet#process()
     */
    @Inject
    @BatchProperty(name = "SYSIN")
    String sysin;
    public String process() {

         AccessMethodServices ams = new AccessMethodServices();

         JsonReader reader = Json.createReader(new StringReader(sysin));
         JsonObject sysin = reader.readObject();
        reader.close();
        JsonArray statements = sysin.getJsonArray("statements");
```

```
          for (JsonValue jsonValue : statements) {
           String statement = jsonValue.toString();
           // Peel off leading and trailing double-quote
           String quotelessStatement = statement.substring(1, statement.length()-
1);
                ams.addInputLine(quotelessStatement);
          }

           int rc = ams.execute();

           String out = ams.getOutputLines();
           log.log(Level.INFO, out);

           String exitStatus = new Integer(rc).toString();
           jobCtx.setExitStatus(exitStatus);
           return exitStatus;

     }

}
```

Document change history

Check the date in the footer of the document for the version of the document.

April 12, 2016	Initial Version
May 20, 2016	Add document number

End of WP102636