WebSphere Application Server

WebSphere Liberty Batch Using DFSort

This document can be found on the web at:
www.ibm.com/support/techdocs
Search for document number **WP102636** under the category of "White Papers"

Version Date: May 20, 2016
See "Document change history" on page 15 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 DFSORT. This is a very commonly used utility that has a wealth of capabilities. It is very fast and provides an easy way to do all sorts of data manipulation in between job steps that are doing various business processing.

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 DFSORT, how do you move that to Java?

In this paper we will take a very very simple DFSORT example and move it 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 DFSORT 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 DFSORT JCL Job

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

```
//STEP1  EXEC PGM=SORT
//SORTIN DD DSN=USER1.DFSORT.TEST1,DISP=SHR
//SORTOUT DD DSN=USER1.DFSORT.TEST2,DISP=SHR
//SYSOUT DD SYSOUT=*
//SYSIN DD *
 SORT FIELDS=COPY
```

That's it. Let's quickly look it over. The first line is just the step in the job that runs the SORT 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 SORTIN and SORTOUT DD statements are names defined by DFSORT as input and output. They point us to my TEST1 (input) and TEST2 (output) sequential datasets. They are allocated with these attributes:

```
Organization  . . . : PS
Record format . . . : F
Record length . . . : 80
Block size  . . . . : 80
1st extent tracks . : 1
Secondary tracks  . : 1
```

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

The next line in our JCL is SYSOUT which directs any messages written by DFSORT into the job output.

And finally we have the control statements that tell DFSORT what to do. All we want it to do is copy the contents of SORTIN to SORTOUT.

There is a veritable wonderland of other control statements you can specify. There are also a whole host of other DD names you can define. But this is enough to show us how it works.

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.DFSORT">
                <properties >
                      <property name="SORTIN" value="USER1.DFSORT.TEST1"/>
                      <property name="SORTOUT" value="USER1.DFSORT.TEST2"/>
                      <property name="control" value="SORT FIELDS=COPY"/>
                      <property name="DDPrefix" value="ABCD"/>
                </properties>
          </batchlet>
    </step>
```

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

We can see a property called SORTIN that has the same value as the dataset name as the DD statement had in the JCL. Likewise the SORTOUT property has the same value as the dataset name in the SORTOUT DD. One thing we are missing that is present in the JCL is the disposition (DISP=SHR). We'll see in a moment that the disposition is hard-coded in the Java code. It could be externalized here if that was important. I was trying to keep things simple.

The next property is the control statements to provide to DFSORT. The value of this property is the same string that was in the JCL SYSIN DD contents.

The final property is something we didn't see in the JCL. This is a value we'll supply to DFSORT to use as the prefix for DD names it looks for. For example, DFSORT normally looks for SORTIN as we saw earlier. We'll use this prefix to cause DFSORT to instead look for ABCDIN. Why would we do that? Well, remember that these jobs are running inside a WebSphere Liberty server. That is a multi-threaded environment. It is possible for multiple jobs to be running at the same time in the same server. And the scope of a DD name is the whole address space. Thus if you had multiple DFSORT jobs running concurrently the DD names could collide and strange things may happen. Using a unique prefix for each job will keep them from colliding. We'll see the code that uses this prefix in a bit.

The Code Behind the JSL

We aren't going to go through all the details of the Java code. 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 those properties from the JSL turn up in the Java code. You do it like this:

```
@Inject
@BatchProperty(name = "SORTIN")
String sortin;

@BatchProperty(name = "SORTOUT")
String sortout;

@BatchProperty(name = "control")
String control;

@BatchProperty(name = "DDPrefix")
String ddPrefix;
```

That incantation sets up four variables: sortin, sortout, control, and ddPrefix. These variables will be automatically given the values specified in the JSL. What if the properties aren'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 DFSORT

JZOS has very nicely provided a Java class that wraps the actual DFSORT 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 a lot of little sample programs that use the DFSORT API. I 'borrowed' from their sample quite a bit 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

Setting Up to Use the DfSort API

Before we can get going we need to do some basic stuff to set up the use of DFSORT from Java. Here's the code fragment:

```
    DfSort dfSort = new DfSort();
    dfSort.setLoggingLevel(ZUtil.LOG_INFO);
    dfSort.setSameAddressSpace(true);
```

We create an instance of the DfSort object (watch the camel-case) that will contain everything about our use of the API.

We set the logging level to “INFO”. This will make it generate the output we would normally expect from DFSORT.

And we set a boolean property that tells JZOS to invoke DFSORT right here inside the address space (the Liberty server) where the Liberty Batch job is running. Setting this to false will cause the spawn of another process in another address where the sort will run.

Setting SORTIN, SORTOUT, ddPrefix, and the Control Statements

Now we need to tell DfSort about our datasets and our control statements that we got from the JSL properties. That's done in just four lines:

```
dfSort.addAllocation("alloc fi("+ddPrefix+"in) da("+sortin+") reuse shr
msg(2)");

dfSort.addAllocation("alloc fi("+ddPrefix+"out) da("+sortout+") reuse shr
msg(2)");

dfSort.addControlStatement("OPTION SORTDD="+ddPrefix);

dfSort.addControlStatement(control);
```

The *addAllocation* method lets you add a dataset allocation to be used by DfSort to invoke the DFSORT utility. The *fi* parameter is the DD name that DFSORT will see. Here we've taken the ddPrefix from the JSL and tucked it in front of the standard DFSORT DD name. The rest is the allocation of the dataset. This is done using the BPXWDYN syntax which you can find here:

https://www.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.bpxb600/rda.htm

As you can see, we substitute in the property values using the sortin/sortout variables whose values got set from the JSL. If you recall, earlier we mentioned that our original JCL specified DISP=SHR but our JSL doesn't set that. That's because in the allocation we have here we specify the *shr* keyword. You could make that into another property if you wanted. In fact, you could make the entire allocation string into a property and not just the dataset name itself.

We also set control statements. We force the SORTDD OPTION to the ddPrefix provided in the JSL and then we also set whatever control statements you provided for us to execute.

Do It!

Time to invoke DFSORT and make things happen. Given all the setup, actually starting the process is easy:

```
dfSort.execute();
```

Getting the Return Code

Next we need to find out what happened. The DfSort API provides a service we can call to get the resulting return code, *getReturnCode*.

```
int rc =0;
try {
   rc = dfSort.getReturnCode();
} catch (RcException rce) {
   log.log(Level.INFO,"Caught RcException: " + rce.getMessage());
   rc = 32;
}
```

It is important to call this because the DFSORT processing is happening on another thread. Calling *getReturnCode* blocks the thread of execution for our batchlet (and job) until the DFSORT processing completes. If something very bad happens to that processing it can result in a Java exception being thrown at our code which is covered by the catch( ) block you see here.

DFSORT defines return code values of 0, 4, 16, 20, 24, and 28. So we set our return code to 32 to indicate something really bad happened that isn't covered by one of the DFSORT return codes. The message we log from the catch path will hopefully tell us something about what went wrong.

Logging Messages

When DFSORT is run from JCL the SYSOUT DD contains messages telling you about what the utility did. Running under the JZOS DfSort 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 DfSort object and logs them.

```
List stderrLines = dfSort.getStderrLines();
for (Iterator i=stderrLines.iterator(); i.hasNext(); ) {
   log.log(Level.INFO, (String)i.next());
}
```

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 DFSORT runs from JCL it writes messages into the SYSOUT of the job. When our JSL runs we saw the code that gets the DFSORT messages from the DfSort 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 DFSORT. It line wraps here a bit, but this is the output from one of the runs. Can you tell which one?

```
ICE201I 0 RECORD TYPE IS F - DATA STARTS IN POSITION 1
ICE751I 0 C5-I21470 C6-BASE   C7-I31998 C8-I25275 C9-BASE   E5-I25610 E7-I31998
ICE143I 0 BLOCKSET     COPY  TECHNIQUE SELECTED
ICE250I 0 VISIT http://www.ibm.com/storage/dfsort FOR DFSORT PAPERS, EXAMPLES AND MORE
ICE000I 0 - CONTROL STATEMENTS FOR 5650-ZOS, Z/OS DFSORT V2R1  - 09:20 ON MON APR 11,
2016 -
          OPTION SORTDD=ABCD
          SORT FIELDS=COPY
ICE193I 0 ICEAM2 INVOCATION ENVIRONMENT IN EFFECT - ICEAM2 ENVIRONMENT SELECTED
ICE088I 0 USER12  .*OMVSEX .        , INPUT LRECL = 80, BLKSIZE = 80, TYPE = F
ICE093I 0 MAIN STORAGE = (MAX,6291456,6291456)
ICE156I 0 MAIN STORAGE ABOVE 16MB = (6234096,6234096)
ICE127I 0 OPTIONS: OVFLO=RC0 ,PAD=RC0 ,TRUNC=RC0
,SPANINC=RC16,VLSCMP=N,SZERO=Y,RESET=Y,VSAMEMT=Y,DYNSPC=256
ICE128I 0 OPTIONS: SIZE=6291456,MAXLIM=1048576,MINLIM=450560,EQUALS=N,LIST=Y,ERET=RC16
,MSGDDN=SYSOUT
ICE129I 0 OPTIONS: VIO=N,RESDNT=ALL ,SMF=NO
,WRKSEC=Y,OUTSEC=Y,VERIFY=N,CHALT=N,DYNALOC=N             ,ABCODE=MSG
ICE130I 0 OPTIONS: RESALL=4096,RESINV=0,SVC=109
,CHECK=Y,WRKREL=Y,OUTREL=Y,CKPT=N,COBEXIT=COB2
ICE131I 0 OPTIONS: TMAXLIM=6291456,ARESALL=0,ARESINV=0,OVERRGN=16384,CINV=Y,CFW=Y,DSA=0
ICE132I 0 OPTIONS: VLSHRT=N,ZDPRINT=Y,IEXIT=N,TEXIT=N,LISTX=N,EFS=NONE
,EXITCK=S,PARMDDN=DFSPARM ,FSZEST=N
ICE133I 0 OPTIONS: HIPRMAX=OPTIMAL,DSPSIZE=MAX
,ODMAXBF=0,SOLRF=Y,VLLONG=N,VSAMIO=N,MOSIZE=MAX
ICE235I 0 OPTIONS: NULLOUT=RC0
ICE236I 0 OPTIONS: DYNAPCT=10 ,MOWRK=Y,TUNE=STOR,EXPMAX=MAX    ,EXPOLD=50%    ,EXPRES=10%
ICE084I 0 EXCP ACCESS METHOD USED FOR ABCDOUT
ICE084I 0 EXCP ACCESS METHOD USED FOR ABCDIN
ICE751I 1 F4-BASE   F5-BASE   E8-I29717
ICE090I 0 OUTPUT LRECL = 80, BLKSIZE = 80, TYPE = F
ICE055I 0 INSERT 0, DELETE 0
ICE054I 0 RECORDS - IN: 16, OUT: 16
ICE052I 0 END OF DFSORT
```

This is the output from the Liberty Batch JSL job run. There are a few hints. The first is message ICE193I. That tells us that ICEAM2 ran. This is a batch program invoked environment. When run directly from JCL the invocation is for ICEAM1.

There is also a difference in ICE088I. In the JCL invocation this shows the job and step from the JCL that called DFSORT. In this listing it shows the address space name (USS address space '3' for userid USER1, yielding USER13) and the step, OMVSEX.

It also turns out that the OVERRGN value reported in ICE131I is different for a shell environment than for a batch environment on my system.

And, of course, the setting of the SORTDD=ABCD option is a dead giveaway.

But Wait...There's More!

I left out a couple of messages that appear in the job log that don't come from DFSORT. They come from the JZOS DfSort interface. They issue a message just as they call DFSORT and just after it returns. The before message isn't that interesting:

```
JVMJZTK2004N Log level has been set to: I
```

But the message issued at the end (after ICE052I above) is very interesting:

```
JVMJZTK2999I jdfsort elapsed time=0.117416 seconds, cpu time=0.005621
seconds
```

Look at that. They very nicely give us both the elapsed and CPU time spent doing the DFSORT processing. If you run DFSORT from regular JCL the messages in the job output will show you the elapsed and CPU time for the step and here we have the same information in the job log from a Liberty Batch step that does the same work.

Enhancements You Could Make

As we said at the beginning of this paper, our intent here was to just show that DFSORT 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 DFSORT. What would you need to do to our example to get there?

To start out you would probably want to add support for more DD statements. DFSORT supports a whole list of standard DDs and it would be simple enough to add those into the JSL step and add a few lines of code to the Java program to @Inject the values and then call *addAllocation* to pass them along to DfSort.

Another important enhancement would be to let the ddPrefix get passed in by the submitter of the job rather than hard code it in the JSL. Making sure the values are different for concurrently running jobs is important.

You could also make the DD support more generic by supporting more than just the dataset names by either adding more properties or turn the one DD property into the actual allocation string.

Furthermore, looking at the JSL we can see these values are all hard-coded. JSL supports a syntax that allows property values to have default values but be over-ridden by parameters supplied when the job is submitted.

Finally, the Java code could do more error checking. It makes no attempt to verify that valid values (or any values at all) were provided for the properties. 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'.

Other SORT products and Utilities

One last thing... DFSORT isn't the only sort product for z/OS. You might have some other competing product. Could you use this same technique to wrap calls to some other sort product or other similar utility? Maybe. Consult with the vendor who sold you the sort product. For other utilities you'd need to find out if/how it can be called from Java.

I would also look closely at the JZOS ZFile class which let's you do a surprising amount of stuff with traditional z/OS datasets (including VSAM). It supports calling BPXWDYN using the same syntax we saw above defining the dataset allocations to DfSort. And there are several samples using ZFile in the samples zip we linked to earlier. You might be able to use that to substitute for things you would do in JCL with DD statements on an IEFBR14 step.

Appendix: Full Source Listing

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

import java.util.Iterator;
import java.util.List;
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 com.ibm.jzos.DfSort;
import com.ibm.jzos.RcException;
import com.ibm.jzos.ZUtil;

public class DFSORT implements Batchlet {

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

   @Inject JobContext jobCtx;

    /**
     * Default constructor.
     */
    public DFSORT() {
    }

   /**
     * @see Batchlet#stop()
     */
    public void stop() {
    }

   /**
     * @see Batchlet#process()
     */
    @Inject
    @BatchProperty(name = "SORTIN")
    String sortin;
    @BatchProperty(name = "SORTOUT")
    String sortout;
    @BatchProperty(name = "control")
    String control;
    @BatchProperty(name = "DDPrefix")
    String ddPrefix;
    public String process() {
         DfSort dfSort = new DfSort();
         dfSort.setLoggingLevel(ZUtil.LOG_INFO);

         //Direct DFSORT to get its input (SORTIN) from the supplied dataset.
         dfSort.addAllocation("alloc fi("+ddPrefix+"in) da("+sortin+") reuse shr
msg(2)");
```

```
        //Direct DFSORT to send its output (SORTOUT) to the supplied DSN.
        dfSort.addAllocation("alloc fi("+ddPrefix+"out) da("+sortout+") reuse
shr msg(2)");

        dfSort.addControlStatement("OPTION SORTDD="+ddPrefix);
        dfSort.addControlStatement(control);

        dfSort.setSameAddressSpace(true);

        dfSort.execute();

        //Wait for dfSort to finish and check the result
        int rc =0;
        try {
              rc = dfSort.getReturnCode();
        } catch (RcException rce) {
              log.log(Level.INFO,"Caught RcException: " + rce.getMessage());
              rc = 32;
        }

        List stderrLines = dfSort.getStderrLines();
        for (Iterator i=stderrLines.iterator(); i.hasNext(); ) {
              log.log(Level.INFO, (String)i.next());
        }

        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