WebSphere Application Server

Monitoring WebSphere Liberty Batch with Batch Events

This document can be found on the web at:
www.ibm.com/support/techdocs
Search for document number **WP102603** under the category of "White Papers"

Version Date: March 11, 2016
See "Document change history" on page 18 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, especially Thai La for churning out all the batch event code way faster than I thought possible.

Table of Contents

Introduction

Starting with the 8.5.5.6 maintenance level, WebSphere Liberty supports the Java EE7 standard. Part of that standard is JSR-352, the open standard for Java Batch. In level 8.5.5.7 support was added to publish batch 'events' at significant points in the life of a batch job running in WebSphere Liberty.

In this paper we'll take a look at the events that get published and the information they provide. We'll take a look at sample code to create a crude monitor that you can use as a starting point for your own monitoring code.

What Are Batch Events?

WebSphere Liberty Batch uses the messaging pub/sub model to publish messages at significant points in the lifecycle of a batch job. These messages are published to a topic tree that represents the various states. All jobs that pass through a given state, for example step completion, will publish a message to the step-completion topic. The message contains enough information to identify the job being run and the step which completed, along with other interesting information.

The entire topic tree can be found in the Knowledge Center article on batch events which is here:

`http://www-01.ibm.com/support/knowledgecenter/SS7K4U_8.5.5/com.ibm.websphere.wlp.nd.multiplatform.doc/ae/twlp_batch_monitoring.html`

(or just search for `twlp_batch_monitoring` in the Knowledge Center)

A subscriber to topics in this tree can thus be notified whenever that event occurs. Or you can subscribe to a higher topic in the tree and, with proper wild-carding, be notified of any event that occurs under that topic. For example, you could subscribe to batch/jobs/execution/# and be notified of any executions that start, restart, etc. as well as the steps within the execution.

Subscribing to a topic only gives you notification of events that occur after you start subscribing. You can't go backwards in time and find out things that happened before. So you may be notified about jobs that completed that you never saw start if it started before you began your subscription.

Batch event messages also have certain properties set that help you identify the job, the execution, and the step. In later sections we'll go through some of the more significant events and look at what information they provide.

Configuring Your Server to Generate Them

Like most things in Liberty, you only get the function if you ask for it. Before we start, we're going to assume that you already have a server configured for Java Batch work. You can find a nice step by step guide to doing that here:

http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102544

That paper is written for z/OS, but most of the material is common to all platforms.

In order to get the server to generate events you need to do three things:

1) Enable the batchManagement feature

   You probably already have this if you've followed the step-by-step guide. You need it to enable the REST interface among other things. But if you haven't done it yet, in the feature manager section of your server configuration just add:

```
feature>batchManagement-1.0</feature>
```

2) Enable the JMS client feature

   You'll need to add this too so the batch code can use JMS to publish messages.

   ```
   <feature>wmqJmsClient-2.0</feature>
   ```

   Or if you're using SIBus for messaging add:

   ```
   <feature>wasJmsClient-2.0</feature>
   ```

3) Configure the server to generate events

   We need a configuration element that tells the server we want batch events. Its called batchJmsEvents.

   ```
   <batchJmsEvents />
   ```

4) Tell the batch event code where to put the messages

   If you've configured your server as a dispatcher (batchJmsDispatcher) then you likely already have a jmsConnectionFactory and the .rar file reference set up. If you don't then you might need to set it up.

   If you include the batchJmsEvents element as we've shown above, it will cause the server to go looking for another configuration element with the id 'batchConnectionFactory'. This points us to either SIBus or MQ. There are samples provided in the Knowledge Center article mentioned earlier.

   You also need to set a variable to tell the server where to find the MQ .rar file if you're using MQ.

   For the system where I ran the jobs and the monitor code we'll go through later, this is the batchConnectionFactory definition I used:

```
<jmsConnectionFactory id="batchConnectionFactory"
jndiName="jms/batch/connectionFactory">
<properties.wmqJms
hostName="wg31.washington.ibm.com"
transportType="CLIENT"
channel="SYSTEM.DEF.SVRCONN"
port="1414"
queueManager="MQS1">
</properties.wmqJms>
</jmsConnectionFactory>

<variable name="wmqJmsClient.rar.location"
value="${server.config.dir}/wmq.jmsra.rar" />
```

Batch Events – A Look Inside

There are a lot of events. We'll take a look at a few of them and what's inside. Before we look inside the messages, let's have a look at some properties of the messages themselves.

Message Properties

A message can carry properties along with the actual body of the message. You can use APIs to get the list of properties and get their values. There are quite a few that are set by the messaging engine itself and a few more that are set by the batch code.

What Event IS this?

If you only subscribe to one specific topic in the tree then when you get notified about a message you'll know exactly which topic occurred. If you're only interested in jobs executions that fail, subscribe to batch/jobs/execution/failed. Then that's all you'll see.

But if you want to monitor multiple events to watch the lifecycle of a job you need a way to figure out which event just happened. Fortunately there's a property on the message that tells you the topic where the message was published. That property is called `JMSDestination`.

Does Anybody Know What Time It Is?

As you watch messages come in for a job you might want to know what time it happened. Fortunately there's another property of the message that gets set when the message is published. The property is called `JMSTimestamp`. That's going to be the time in milliseconds since the epoch which is back in 1970.

Let Me See Some ID...

Job instances, executions, and steps all have generated identifiers. You can use those identifiers to correlate different events together. If you have an event for the beginning of the dispatch of an execution, you can use the execution identifier to recognize the corresponding end of the execution.

Where appropriate the batchManagement code will set properties containing these identifiers to allow you to easily access them. The property names are:

- `com_ibm_ws_batch_internal_jobInstanceId`
- `com_ibm_ws_batch_internal_jobExecutionId`
- `com_ibm_ws_batch_internal_stepExecutionId`

Instances and Executions

If you're not intimately familiar with the JSR-352 specification all this talk of instances and executions may have left you a bit confused. If you're writing a monitor to watch JSR-352 jobs run in Liberty, you'll need to understand the distinction and manage the events appropriately.

Of course the final word on this is in the JSR-352 specification itself, but I'll try to summarize here. Essentially if you submit a job you get an instance of that job. When we try to run it we create an execution. If the job fails and becomes restartable, then you can try to restart the job. Restarting the job creates a new *execution* for the original *instance* of the job.

Every time you START a job, you get a new instance and an execution. Every time you RESTART a job you get a new execution.

You can look at the identifiers we talked about just above to sort this out. Every job you start/submit will get a unique job instance identifier. Every time we try to run it (originally or as a restart) will get

a unique job execution identifier. Each step run during an execution also gets its own unique identifier.

How unique are the identifiers? They are tied to the Job Repository database tables. Every server sharing those tables (if they are shared at all) will generate identifiers unique within that set of servers. Servers using an in-memory Job Repository will generate numbers unique within that server, just like a server with exclusive access to a Job Repository.

Identifiers in the database don't reset unless you wipe out the tables and start over. So they should just grow forever. Identifiers in an in-memory Job Repository will restart every time you start the server.

Job Instance Events

We're not going to go through all the events. See the Knowledge Center article referenced earlier for the complete list. I just want to review the ones you're likely to see in normal (successful) processing.

In a single server configuration you'll see three Job Instance events published for the normal processing of a job. These are:

- submitted
- dispatched
- completed

If you are running in a multi-JVM configuration then we publish two more events to let you see the events around the message queue between the dispatcher and an executor. You'll get events for:

- jms_queued
- jms_consumed

They will be issued in that order, between the submitted and dispatched events.

You can take the timestamp on the consumed message and subtract the timestamp on the queued message to find out how long the job waiting in the queue. You can subtract the dispatched timestamp from the completed timestamp to find out how long the job took to run.

That's all from the message properties. What's in the message itself? Let's take a look.

Submitted

Here's the JSON string that comes from a Job Instance Submitted message:

```
{"jobName":"(NOT SET)","instanceId":3,"appName":"SleepyBatchletSample-
1.0#SleepyBatchletSample-
1.0.war","submitter":"Fred","batchStatus":"STARTING","jobXMLName":"sleepy
-batchlet.xml","instanceState":"SUBMITTED"}
```

Let's pull that apart and see what we've got.

"jobName":"(NOT SET)"

This comes from the id value in the <job> element of the JSL. We haven't actually read the JSL yet so we don't know what it is yet.

"instanceId":3

This is the Job Instance ID we talked about earlier. It will match the Job Instance property value.

"appName":"SleepyBatchletSample-1.0#SleepyBatchletSample-1.0.war"

This is the full application name derived from the applicationName parameter used to submit the job via the batchManager utility.

"submitter":"Fred"

The userid that submitted the job.

"batchStatus":"STARTING"

The status of the job. Batch Status is a term defined by the JSR-352 specification and this field represents a state as defined by the specification (see section 8.7). This is not the same as the Exit Status of the job.

"jobXMLName":"sleepy-batchlet.xml"

This is the name of the file inside the batch application that contains the JSL used to execute the job. The file name is specified by the jobXMLName parameter on the batchManager utility.

"instanceState":"SUBMITTED"

This is the state of the Job Instance.

Dispatched

Here's the JSON string for a Job Instance Dispatched event:

```
{"jobName":"sleepy-
batchlet","instanceId":3,"appName":"SleepyBatchletSample-
1.0#SleepyBatchletSample-
1.0.war","submitter":"Fred","batchStatus":"STARTED","jobXMLName":"sleepy-
batchlet.xml","instanceState":"DISPATCHED"}
```

As you can see, a lot of this is exactly the same as the Submitted event. Notice that the jobName is now set because we've read the JSL. And the batchStatus has changed to STARTED while the instanceState is now DISPATCHED. Otherwise its the same as before.

Completed

Finally, here's the JSON string for a Job Instance Completed:


```
{"jobName":"sleepy-
batchlet","instanceId":3,"appName":"SleepyBatchletSample-
1.0#SleepyBatchletSample-
1.0.war","submitter":"Fred","batchStatus":"COMPLETED","jobXMLName":"sleep
y-batchlet.xml","instanceState":"COMPLETED"}
```


Again, pretty much the same except that our two status fields now both say COMPLETED.

Job Execution Events

As discussed earlier, a Job Execution represents an attempt to run a Job Instance. As such its a bit closer to the details of what's going on and there are both more messages and more detail. First let's look at the main messages we get for a normal execution of a single step job. As before, there are other messages related to the execution. We'll just take a peek at a few.

For a simple job you will see:

- Starting
- Started
- Step/Started
- Step/Completed
- Completed

First off, what's the difference between Starting and Started? This goes back to the batch Status we talked about under Job Instance Events. Starting is a batch status that means the batch infrastructure has the job and is starting to process it. A state of Started means that actual execution of the job has begun (or will as soon as we're done publishing this event :-)

Note also that we have Started/Completed states for the Step in our job. In a multi-step job you will see events for each step. Remember that steps have identifiers that are available as properties on the message so you can easily match up starting/ending events for a particular step.

Remember also that these messages will show up inside of the Job Instance events. For example, the Instance will publish a Dispatched event before the Execution publishes its Started event.

But the really cool stuff is in the message content. We're just going to look at two samples, the Step/Completed event and the Job Execution Completed event. That's because these have the most interesting stuff filled in. If you look closely you'll notice that the samples I'm using here are from a different job than the Job Instance samples I used earlier. This job had checkpoints so it has more interesting data than SleepyBatchlet.

The Step/Completed message looks like this:


```
{"stepExecutionId":90,"stepName":"addBonus","executionId":87,"instanceId"
:87,"batchStatus":"COMPLETED","startTime":"2015/11/12 13:02:03.399
+0000","endTime":"2015/11/12 13:02:03.966
+0000","exitStatus":"COMPLETED","metrics":
{"READ_COUNT":"1000","WRITE_COUNT":"1000","COMMIT_COUNT":"11","ROLLBACK_C
```

OUNT":"0","READ_SKIP_COUNT":"0","PROCESS_SKIP_COUNT":"0","FILTER_COUNT":"
0","WRITE_SKIP_COUNT":"0"}}

So let's go through the pieces..

"stepExecutionId":90

As we mentioned earlier, each step gets a unique identifier. This is the identifier for this step. Its also available as a property on the message.

"stepName":"addBonus"

This comes from the 'id' for the step in the JSL. For a multi-step job this allows you to identify which step you're on.

"executionId":87

As we've discussed before, this is the execution identifier.

"instanceId":87

And this is the job instance identifier.

"batchStatus":"COMPLETED"

And, again, this is the batch status we've talked about before.

"startTime":"2015/11/12 13:02:03.399 +0000"

New stuff! This is the time when the step started. It is in the message as a string so its tough to do math with, but handy if you're creating a log. If you want to do math, get the time from the message property on the step started/completed events.

"endTime":"2015/11/12 13:02:03.966 +0000"

And this is the time the step ended. Again, its just a string.

"exitStatus":"COMPLETED"

This is the exit status for the step. Its a string set by the application itself.

"metrics":

The JSR-352 specification defines a set of metrics that can be retrieved by the application. These are the metrics for the step. They are contained in a JSON array inside the 'metrics' value. Details about these values can be found in the specification, but we'll skim through them here.

"READ_COUNT":"1000"

How many times did we call the 'reader' for this chunk step?

"WRITE_COUNT":"1000"

How many things did the writer write for this chunk step? Remember this isn't how many times it got called because we only call the writer at checkpoints. Each time it gets a list of things to write. This is the count of things it got passed.

"COMMIT_COUNT":"11"

This is how many times we had a checkpoint (and how many times we called the writer).

"ROLLBACK_COUNT":"0"

The number of times we rolled back due to an error.

"READ_SKIP_COUNT":"0"

The number of records skipped due to error detected by the reader.

"PROCESS_SKIP_COUNT":"0"

The number of exceptions thrown by the item processor.

"FILTER_COUNT":"0"

The number of items filtered by the item processor (the processor returned a null instead of an object to be given to the writer).

"WRITE_SKIP_COUNT":"0"

The number of skippable exceptions thrown by the item writer.

The Job Execution Completed message looks like this:


```
{"jobName":"SimpleBonusPayoutJob","executionId":87,"instanceId":87,"batch
Status":"COMPLETED","exitStatus":"COMPLETED","createTime":"2015/11/12
13:02:03.223 +0000","endTime":"2015/11/12 13:02:04.003
+0000","lastUpdatedTime":"2015/11/12 13:02:04.003
+0000","startTime":"2015/11/12 13:02:03.261 +0000","jobParameters":
{"tableName":"BONUS.ACCOUNT","dsJNDI":"jdbc/bonus"},"restUrl":"https://19
2.168.17.215:11001/ibm/api/batch","serverId":"localhost//shared/jsrhome/l
iberty/jsrexec1","logpath":"/shared/jsrhome/liberty/servers/jsrexec1/logs
/joblogs/SimpleBonusPayoutJob/2015-11-12/instance.87/execution.87/"}
```


And if we take that apart:

"jobName":"SimpleBonusPayoutJob"

As in the job instance messages this is the job name from the 'id' field in the JSL job element.

"executionId":87

Our job execution identifier.

"instanceId":87

Our job instance identifier.

"batchStatus":"COMPLETED"

Our final batch status.

"exitStatus":"COMPLETED"

Our final exit status, as set by the application.

"createTime":"2015/11/12 13:02:03.223 +0000"

The time the job was created, as a string.

"endTime":"2015/11/12 13:02:04.003 +0000"

The time the job ended, as a string. Again, easier to do math with the timestamp property from the various job event messages than these strings.

"lastUpdatedTime":"2015/11/12 13:02:04.003 +0000"

The last time the job information in the Repository was updated.

"startTime":"2015/11/12 13:02:03.261 +0000"

Weirdly last in the timestamps in the message, we have the time execution started.

"jobParameters":{"tableName":"BONUS.ACCOUNT","dsJNDI":"jdbc/bonus"}

This is fairly cool. We can see the parameters that were used to submit the job. These aren't job properties contained inside the JSL, but parameters passed in which were probably used to set values to for job properties.

"restUrl":"https://192.168.17.215:11001/ibm/api/batch"

If you need to contact the server that ran the job using the REST API, this is the URL to use. If your monitor wanted to retrieve the joblog for the job, this is where you would go to get it.

"serverId":"localhost//shared/jsrhome/liberty/jsrexec1"

The 'name' of the server that ran the job. Its essentially the location of the server.xml file for the server.

```
"logpath":"/shared/jsrhome/liberty/servers/jsrexec1/logs/joblogs/SimpleBonusPayoutJob/2015-11-12/instance.87/execution.87/"
```

If you don't want to use the REST API to fetch the joblog, you could also go directly to the system than ran the job and go to this path in the file system and retrieve it. Or if the server's REST API is unusable (because you don't have access to it, or maybe because the server is down).

So what? What could you do with this data?

I've tried to give you a general sense of the sort of data that's available to you through the events published by the WebSphere batch runtime. I thought I would use this section to just dream up some different things you might be able to do....

Basic Automation – you could write code to watch for job/execution/failed messages and restart the job (or trigger existing automation that does that).

Simple Chargeback – Record jobs submitted and the user that submitted them to produce reports.

Basic Alerts – By tracking submitted jobs against dispatched and completed jobs you can calculate how many jobs are in the system but aren't actually running anywhere. If that number grows something is wrong. You could also build up some history of how long certain jobs usually take and raise an alert if you notice a job dispatch but not end in a 'reasonable' amount of time (or if a particular step takes longer than usual)

Progress Monitor – For a chunk step you'll get events for every checkpoint. You could track how much time passes between checkpoints. If you know how many checkpoints there will be, then you could provide a constantly adjusted guess about when the step will complete.

What else?

Writing Your Own Monitor

There are numerous ways to handle messages published to a topic. Probably the easiest thing to do is write an MDB (Message Driven Bean) and configure a WebSphere server to subscribe to the topic and run the MDB.

But I wanted to write a stand-alone monitor that would subscribe to the whole topic tree and spew out messages as the events came in. Then I can submit jobs and watch the messages scroll past on a window on my laptop.

There are several different ways to do that. I chose to use the MQ Java APIs (not the JMS ones, but the MQ specific ones). Why? Well...to be honest Google led me there first.

Furthermore...I'm not an MQ expert. The best I can say about this example is that it works. If you're familiar with writing applications that interact with MQ I'm sure there are comments to be made. But it works...(or at least it works for me :-)

With all that said, I'll take you on a quick stroll through the code. The whole source should be attached to the same web site where you found this PDF.

Connecting to MQ

There are several ways to connect to MQ. I wrote this to support two of them. The easy way is to run locally to MQ and just specify the queue manager name. If you're remote from MQ (like I was

on my laptop) then you'll need the host/port for MQ along with the channel name. All these are things you can find from the jmsConnectionFactory we put in the Liberty server.xml way back at the top of this paper.

If you are using the host/port/channel then you'll need to set those into something called the MQEnvironment. Like this (where you've gotten the values from input parms or properties)

```
MQEnvironment.hostname = new String(hostname);
MQEnvironment.port = new Integer(port).intValue();
MQEnvironment.channel = new String(channel);
```

And then you create an MQQueueManager. Do this specifying the queue manager name (qmgr here). If you've set up things in the MQEnvironment it just finds them.

```
MQQueueManager mqqm = new MQQueueManager(qmgr);
```

Subscribe to the Topic

Now we need to subscribe to the topic we're interested in. We set up some options first. Honestly I stole these options from an example and they seemed to work. MQ has a lot of options.

```
int options =   CMQC.MQSO_CREATE|
                CMQC.MQSO_MANAGED|
                CMQC.MQSO_NON_DURABLE|
                CMQC.MQSO_FAIL_IF_QUIESCING|
                CMQC.MQSO_WILDCARD_TOPIC;
```

Next we set up the topic we want to subscribe to. The topic comes in two parts, the name and the object. MQ runs those together to form the whole string. I just put all of it in one and left the other one as an empty string. I'm sure its like this for some cool reason that I'm failing to take advantage of.

```
String topicName = new String("batch/#");
String topicObject = new String("");
```

And now we subscribe to the topic:

```
MQTopic mqt = mqqm.accessTopic(  topicName,
                                 topicObject,
                                 CMQC.MQTOPIC_OPEN_AS_SUBSCRIPTION,
                                 options);
```

Getting A Message

At this point you're ready to start getting messages. You might want to do this in a loop. In my little monitor I just went into a loop forever. You may want a way out.

To get a message you'll need to create a new MQMessage object and set it up. Since we're happy to take any message and aren't waiting for something specific tell it you don't want a specific message or correlation ID. And a couple of other options involving character encoding.

```
MQMessage message = new MQMessage();
message.messageId = CMQC.MQMI_NONE;
message.correlationId = CMQC.MQCI_NONE;
message.encoding = CMQC.MQENC_NATIVE;
message.characterSet = CMQC.MQCCSI_Q_MGR;
```

And then create an MQGetMessageOptions object and set some more options. I wanted to wait forever for messages. You might not want to.

```
MQGetMessageOptions messageOptions = new MQGetMessageOptions();
messageOptions.options=CMQC.MQGMO_WAIT|CMQC.MQGMO_CONVERT;
messageOptions.waitInterval=CMQC.MQWI_UNLIMITED;
```

Let's get a message..

```
mqt.get(message, messageOptions);
```

Processing the Message

Alright...we got something..let's find out what topic it came from:

```
String messageTopic = message.getStringProperty("JMSDestination");
```

And pick up the time the event happened and convert it to something printable:

```
Long timestamp = message.getInt8Property("JMSTimestamp");
Date date = new Date(timestamp);
String sDate = new SimpleDateFormat
                        ("yyyy-MM-dd HH:mm:ss.SSS").format(date);
```

We'll pick up the job instance ID from its property:

```
Long jobInstanceId = message.getInt8Property(
                       "com_ibm_ws_batch_internal_jobInstanceId");
```

The execution and step identifiers might not be set if this is a job instance event. We could look at the topic and figure it out. I decided instead to just look at the properties provided and see if it was there. You can call the getPropertyNames API to get back a list of properties. You can specify a pattern (or specific name) of the property you're interested in. Regardless how you do it, you get

back an enumeration of the properties that match, if there are any. So I just asked for an enumeration of the specific property I wanted and if I got back an enumeration with any content, it was there. There's probably a better way, but this worked.

First, set up the enumeration and the variables to hold the identifiers. We'll initialize the identifiers to zero which is what will show up if the properties aren't set.

```
Enumeration<String> en;
Long jobExecutionId = 0L;
Long stepExecutionId = 0L;
```

Now look for each property and get its value if it exists in this message.

```
en = message.getPropertyNames(
                    "com_ibm_ws_batch_internal_jobExecutionId");
if (en.hasMoreElements()) {
   jobExecutionId = message.getInt8Property(
                    "com_ibm_ws_batch_internal_jobExecutionId");
}

en = message.getPropertyNames(
                    "com_ibm_ws_batch_internal_stepExecutionId");
if (en.hasMoreElements()) {
   stepExecutionId = message.getInt8Property(
                    "com_ibm_ws_batch_internal_stepExecutionId");
}
```

And finally let's go get the actual message which is a big JSON string. We need to find out how long it is so we can tell the 'read' API how much we want (all of it).

```
int textLength = message.getMessageLength();
String text = message.readStringOfCharLength(textLength);
```

At this point I just printed out everything I had. The date/time, the three identifiers (or zeroes), and then the JSON string from the message. You could use a JSON parser at this point to pull apart the message and pick out the information you care about.

```
    System.out.println(sDate+"->" + messageTopic + "(" + jobInstanceId +
":" + jobExecutionId + ":" + stepExecutionId +"): " +text);
```

Finishing Up

Finally, you should close your connection as you exit (if you do).

```
mqt.close();
```

Building and Running the Code

If you want to run the sample that's attached or write and run your own monitoring code that uses the Java MQ APIs you're going to need a few things.

To get it to build, you're going to need the MQ Java library .jar files. These are probably in the java/lib directory of wherever MQ got installed. MQ STRONGLY discourages you from making copies of these because you'll have problems keeping track of maintenance. You're supposed to build against (and run against) the licensed copy, wherever that is. So there you go.

I needed three .jar files:

- com.ibm.mq.headers.jar
- com.ibm.mq.jmqi.jar
- com.ibm.mq.mq.jar

And when you run you'll need to make sure those .jar files are on on your CLASSPATH. When running on z/OS I wound up putting the whole MQ java/lib on my LIBPATH, and setting a STEPLIB to the SCSQAUTH dataset. Depending on your system setup you might not need to do that. Running from Eclipse on my laptop just needed those .jar files in the build path and I was all set. Running from a shell on z/OS needed a bit more, but perhaps that's just because of how things were set up there.

Best of luck!

Document change history

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

November 30, 2015	Initial Version
March 11, 2016	Corrected document title in document properties

End of WP102603