WebSphere Application Server

Creating a 'Job Class' for WebSphere Liberty Batch

This document can be found on the web at:
www.ibm.com/support/techdocs
Search for document number **WP102600** under the category of "White Papers"

Version Date: November 23, 2015
See "Document change history" on page 33 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

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. The core support for the standard allows you to use the spec-defined JobOperator API to submit a job to run right there in the JVM where the JSR is implemented. That's handy if you're running in a JVM started just for the purpose of running this job, but less convenient if you have a server that exists to run various jobs.

It would be nice if there was some external way to reach into the server to submit a job. To allow that IBM built a REST interface in Liberty that lets you interact with the batch feature externally. But that still only lets you run a job in the specific server you contact via REST.

It would be nice to have a whole set of servers configured to run various batch jobs and one (or several) servers you could contact via REST that would cause the job to run wherever it was appropriate in that set of servers. But what controls where it is appropriate to run a job? In traditional z/OS Batch you would define JES initiators in various job classes and when a job is submitted a job class is specified and it runs in an initiator in the right class.

It would be nice if WebSphere Liberty Batch supported something like a JES Job Class. What it actually provides is something more powerful, but less specific.

In this paper we'll go through how all of these pieces work and how you can configure your servers to get what is pretty much the same as a traditional job class.

Let's get started...

An Overview of the Single and Multi-JVM Models

As mentioned in the introduction, the WebSphere Liberty implementation of JSR-352 supports both a single and multi-JVM model. In the single JVM model the implementation of the JSR-352 programming model (sometimes called the batch container) lives in a Liberty server and can run jobs using applications installed in that server. To submit those jobs you need an application installed in that server that calls the JSR-352 defined JobOperator API to start( ) the job.

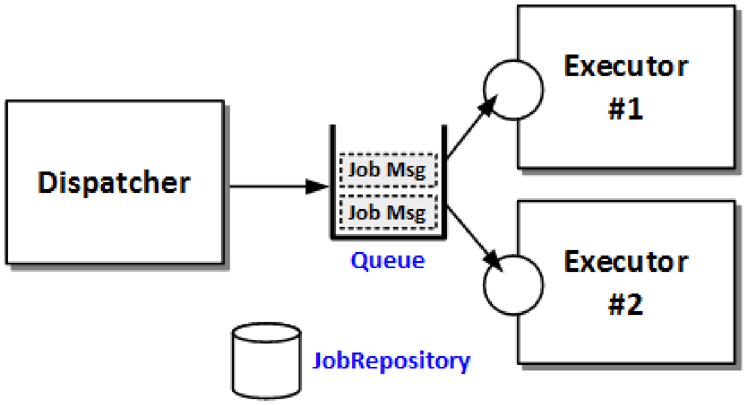
The JSR-352 implementation takes over from there. It finds the JSL located in the application package and begins to execute the steps defined in the JSL. This is a great way to get started using JSR-352 and is how most developers would use it. Unless you're using the very cool support included in the WebSphere Developer Tools. But that's a topic for another paper. In fact, you might take a look here:

https://developer.ibm.com/wasdev/docs/creating-simple-java-batch-application-using-websphere-developer-tools/

But what about a real production environment. For that you'd want multiple application servers able to run different batch applications. You'd want one interface you can use to submit jobs into that set of servers. To support that IBM added some extra capabilities into our implementation of the JSR-352 specification in Liberty.

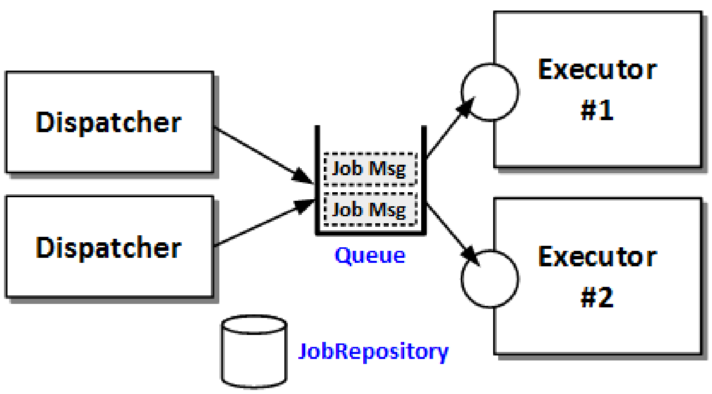
There are two concepts involved. The first is the idea of a dispatcher. By adding some configuration the server changes its behavior. Instead of just running batch jobs here in the server, it creates a message that represents the job and puts that message on a queue (either MQ or SIBus). The information about the job itself lives in the JobRepository (where the specification says implementations have to keep such information) just like it does for jobs submitted within a single server. But it is the responsibility of the message to get the job running somewhere.

Which brings us to the second concept. By adding some different configuration to a batch server you can transform it into a batch executor. This is a server that picks up the messages from the queue and executes the jobs they represent. Note that since both dispatcher and executor are involved in the job, the database for the JobRepository has to be accessible to them both.



Setting up a dispatcher and a set of executor servers allows you to contact a single server (the dispatcher) and run jobs across the set of servers. Which servers run which jobs? Ah...hang on. We'll get there.

But wait, having a single dispatcher as a contact point sounds nice, but what about availability? If that server goes down, does that mean you can't submit new jobs or manage any existing jobs? That seems bad. Not a problem. You can define *multiple* dispatcher servers, all using the same Job Repository, configured to put messages into the same queue.



But wait, we said earlier that you use the JobOperator API to submit jobs. If I have multiple servers configured as a dispatcher, do I need an application in each dispatcher server that calls JobOperator? That seems weird...

Actually it's completely wrong. You can only use the JobOperator API to submit a job that lives in the same application as the code calling JobOperator. And so, in a multi-JVM configuration your application doesn't live in both the dispatcher and executor servers. You actually can't call JobOperator to submit a job in a multi-JVM batch environment. You have to use something else.

Read on...

Submitting a Job – The JobOperator, REST, and Command Line Interfaces

We've already talked a lot about the JobOperator interface. But just to summarize here the JobOperator API is part of the JSR-352 specification. It provides a way, from within an application installed in a Liberty server, to submit a job whose pieces are inside that same application. There are also APIs to check the status of the job, try to stop it, etc.

But what we really care about in this section are the cooler options provided in the IBM WebSphere Liberty implementation. The first cooler thing is the REST interface we've provided to let you manage Java Batch jobs remotely from the server that's running them. You don't need pieces inside the batch application to help you submit it. You can have some other piece of code running somewhere else use a REST/JSON interface to get the job running, check on its status, get its output, and purge it when you're done.

If you're reading closely you'll have noticed that our list of 'stuff you can do' for the REST interface went on a bit longer than the list we had for the JobOperator API. That's because there are things you can do from the REST interface that you can't do via JobOperator. And that is because JobOperator is a specification defined interface where the REST interface is an IBM extension.

The REST interface lets you do things like purge a job from the JobRepository. There's no way to do that from JobOperator. (In case you're wondering why...its because the specification doesn't define how the JobRepository is implemented, so it also doesn't define how you get things out of it).

The REST interface also lets you get the job log. A job log is the set of messages written by the job as it executed. The specification doesn't talk at all about where messages a job writes end up. So there's nothing in JobOperator about how to get to them. The concept of a job log is an IBM extension to the specification.

Having a REST/JSON interface to your batch environment allows you write other applications that submit batch jobs as part of their processing. It allows Enterprise Schedulers (such as IBM's Tivoli Workload Scheduler) to interact directly with a Java Batch environment and submit and manage jobs just like any other work they manage.

That's all great. But what if you just have a job you want to submit? What if you want to run the Java Batch job as part of some other job or script that you already have? Do you have to write your own code to drive the REST interface to get the job submitted? Well, you could, but you don't have to. IBM has you covered.

We've also provided not one, but two Command Line Interfaces (CLI hereafter) that wrap the REST interface and let you interact with your Liberty Batch environment from jobs, scripts, or (if you like that sort of thing) an actual command line.

Why two implementations of the CLI? One of them does what we said. It is written in Java and uses the REST interface under the covers to submit jobs and so forth. The other CLI is written in C and compiled to run on z/OS. It uses the z/OS-exclusive WOLA (WebSphere Optimized Local Adapters aka zosLocalAdapters) to communicate with the Liberty server. This avoids the need to start a JVM on z/OS just to submit a job in a JVM that's already running.

In a Multi-JVM Model, What Controls Where a Job Runs?

The short answer is: you do. Let's just start with a simple case with two servers both pulling messages from the job message queue. Given no extra configuration the servers will both just pull off whatever message is next and run it. And, if the servers are the same, that's probably just fine.

But suppose they aren't. Suppose they have different batch applications installed. Well then you'd want to have something that made sure jobs got to the server where the application lived. It won't work very well otherwise. To do that we need to know two things: how to configure the server to only get certain messages and how to identify which messages the server can handle.

The first of those is done using the message selector in the configuration of the executor server. This is a normal part of setting up a server to process messages. As part of configuring the activation specification, you tell the server where the messaging engine is (host/port) and various other things, including something called a message selector.

The message selector is essentially a logical statement that tells the server how to select messages from the queue based on properties of the messages in the queue. For example, if all messages placed on a queue have a property called “stockName” and you want to process messages related to IBM stock, then you would configure a message selector like this:

```
messageSelector="stockName='IBM'"
```

Note the double quotes around the value of 'messageSelector' and the single quotes around the property value to match. You can create complex selectors with ANDs and ORs and other syntax as needed.

Well, so what? As it happens, messages representing jobs are placed on the queue by the dispatcher with a special property that is the application name. The magic property name is com_ibm_ws_batch_applicationName. It's like that to avoid accidentally colliding with other properties you might use. That means you can configure the message selector to only pick up messages that are for jobs involving applications installed on that server. You might set it to something like this:

```
messageSelector="com_ibm_ws_batch_applicationName='SleepyBatchletSample-1.0'"
```

Where does the dispatcher get the value for the application name? You tell it when you submit the job using the REST or Command Line Interface. Using the CLI it is the value you give on the --applicationName parameter.

Therefore, as you install new batch applications into a server which is acting as an executor you'll need to update the selector to also accept messages with the new application name. You can do that with wild-carding if you're careful with your application names, or by providing a list ORed together, like this:

```
messageSelector="com_ibm_ws_batch_applicationName='SleepyBatchletSample-
1.0' OR com_ibm_ws_batch_applicationName='BonusPayout-1.0'"
```

Defining Your Own Properties for Job Selection

That's pretty cool, but how does it help us? Well, the application name is a special message property set by the dispatcher, but you can define your own properties. In fact, any job parameter you specify when you submit the job over the REST interface or using a CLI will end up as a property on the message. Mostly..if your parameter name violates the rules for message property names then we can't use it. The most obvious problem is that dots aren't allowed in message property names. That's why the full application name property has all those underscores.

This is very cool. You can add any name/value pair you like as a parameter when you submit the job and it will end up as part of the message and you can use those property values in conjunction with the messageSelector to control where messages (and jobs) end up.

Consider our stockName example from earlier. If the name of the stock to process was a parameter provided when the job was submitted you could use the messageSelector value we show above to cause jobs relating to IBM stock to only run in that server.

What can you use as a job parameter for message selection? Anything you like. It is just a name/value pair and you get to make up the name(s) and all the values.

That means you could configure your execution servers with message selectors looking for a property called jobClass. Some servers might select messages with `jobClass='FAST'` or maybe with `jobClasss='A'` or whatever values mean something to you. You can even define a server with a selector looking for jobClass IS NULL which becomes the server that gets messages for jobs that were submitted without a value for the jobClass parameter. That server runs jobs in the default job class.

Making “JobClass” Mean Something

Alright, so you've configured a bunch of servers and you've given them messageSelector strings that look for jobClass values of 'A' and 'B'. You've decided that jobs in class A are more important than jobs in class B. How do you make that actually happen?

In a distributed environment it might have to do with the hardware the particular servers are on. Maybe the 'A' servers are running in 16-core machines with lots of memory and the 'B' servers are running on an old Pentium box under somebody's desk :-)

On z/OS we can use WorkLoad Manager (WLM) to manage the servers differently. If our servers are started as Started Tasks then we can use the WLM STC classification rules to assign our two servers to different WLM Service Classes based on the Started Task Name (presumably different for our two servers). These Service Classes would be assigned a goal.

Since WLM doesn't “see” the jobs running inside the server, we need to assign a Velocity Goal to each Service Class. A Velocity Goal is just a percentage where bigger is 'better' (see the WLM documentation for more information about Velocity Goals). We also need to assign an importance to the goal. The importance tells WLM which work to sacrifice if it can't manage system resources to meet all the goals of all the work. Bigger importance numbers mean the work is less important (Importance-1 work gets its goals met before Importance-3 work).

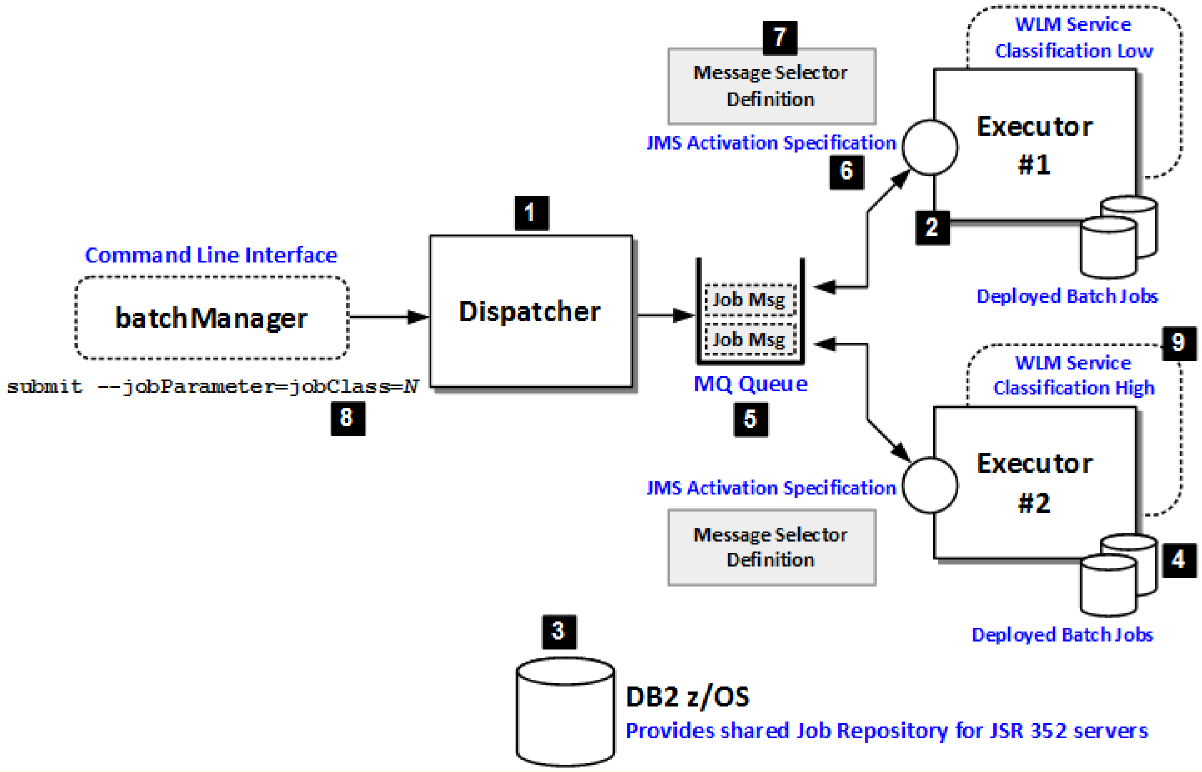
And that's all there is to it. Submit your jobs specifying a jobClass value that matches a value specified in the messageSelector of one of the executor servers. Classify that server to WLM to get system resource in accordance with whatever meaning you have assigned to the jobClass values.

In Practice

What follows is a description and illustration of a real environment we constructed to show submitting jobs with a "Job Class."

Overview

The following picture illustrates the environment we built. The numbered blocks correspond to notes that follow:



The configuration details for all of this are provided in the sections that follow. This is an overview illustration designed to help you see "the big picture."

Notes:

1. The Liberty server configured as the Dispatcher.
2. Two Liberty servers configured as Executors
3. DB2 z/OS served as the JSR 352 Job Repository for this environment.

   The multi-JVM model of Dispatchers and Executors requires that all the servers have access to the same Job Repository information. DB2 LUW would have worked as well. The in-memory option of IBM's WebSphere Java Batch would not work as that is not shareable between servers.

4. The Java Batch jobs were deployed into the Executor servers. For this exercise we used two supplied samples: SleepyBatchlet and BonusPayout.
5. An MQ queue was defined to serve as the exchange point between Dispatcher and Executors.

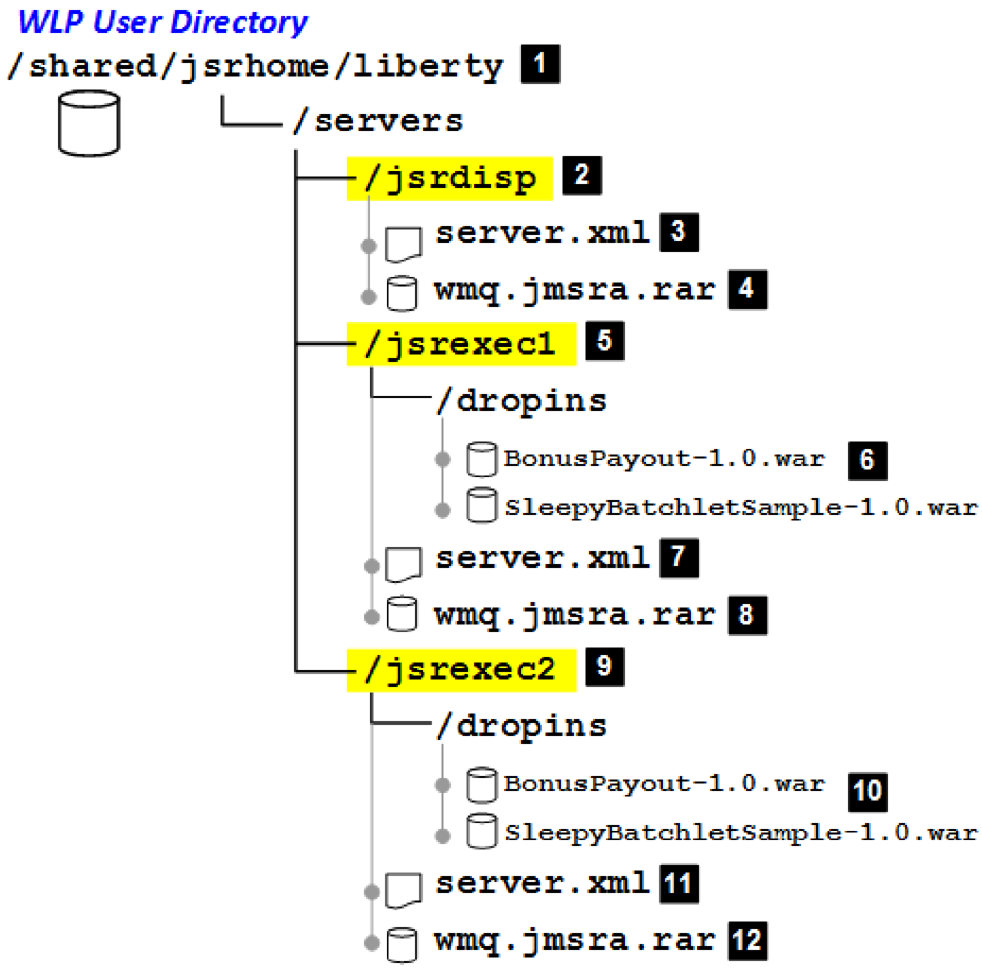
   The default messaging of WebSphere Liberty would work as well. We chose IBM MQ simply because it is very widely used.

6. Each Executor was defined with a JMS activation specification that listened on the MQ queue.

7. Each Executor was defined with a *message selector* that determine what job submission messages were to be picked up by that server.
8. Jobs were submitted using the `batchManager` command line utility. A job parameter of `jobClass=` was passed in as well. This related to the message selector (7) defined in each Executor.
9. Each Executor started task was given a different WLM velocity goal. In this way the jobs that ran in each, based on the jobClass= parameter and the message selector, ran in a server operating with a different WLM classification goal.

Liberty servers and the the configuration directory

The following diagram illustrates the directory structure for the three Liberty servers. The numbered blocks correspond to notes that follow.



Notes:

1. The three Liberty servers were constructed under the same `WLP_USER_DIR` directory. That is *not* a requirement; it is simply how we did it for this illustration.

   The Dispatcher and Executor servers may be under the same `WLP_USER_DIR` (as shown), under different directories on the same server platform, or on completely separate servers.

2. The Dispatcher server was named `jsrdisp` and this was its directory.
3. The Dispatcher's `server.xml` contained the definitions that enabled the dispatcher function and provided details for access to DB2 and MQ. To see details of the Dispatcher's

configuration XML, see "Dispatcher server.xml" starting on page 16. To review the complete `server.xml` for the Dispatcher, go to page 20.

4. The Dispatcher requires access to the queueing mechanism, which in this case was IBM MQ. This file was the MQ resource adapter. JMS definitions in the `server.xml` provided the Dispatcher with information about this RAR, the MQ queue manager, and the queue to use for dispatching jobs.

   Notice that the same RAR file exists for each server. We *could* have employed Liberty's ability to share artifacts between servers. We chose **not** to illustrate that. Why? Because the ability to share artifacts between servers assumes the servers all reside under the same user directory. They did in *this* case, but they may not in yours. Therefore, we illustrated each server having its own complete set of configuration and resource artifacts. If your setup permits sharing and you wish to do that, you certainly can.

5. The first of two Executor servers was named `jsrexec1`. This is its directory.
6. The two sample applications were deployed to the `/dropins` directory of the this executor.

   Again, we could have illustrated sharing of these two applications between the two Executor servers, but we chose not to. So each Executor has the same applications deployed in their respective `/dropins` directories.

7. The first Executor's `server.xml` contained the definitions that enabled the executor function and provided details for access to DB2 and MQ. So see details of this Executor's configuration XML, see "Executor #1 server.xml" starting on page 21. To review the complete `server.xml` for this Executor, go to page 24.

   Executor #2's `server.xml` (11) was very similar to #1's. One difference to look for when you review the XML is the different message selector definitions. That is what determines what job submission messages to pick up. To review that server's XML go to page 26.

8. The MQ resource adapter file used by this Executor[1].
9. The second of two Executor servers was named `jsrexec2`. This is its directory.
10. The two sample applications were deployed to the `/dropins` directory of this executor[2].
11. The second Executor's `server.xml`. This was very similar to first executor's `server.xml`. To see the complete XML for this executor, go to page 26.
12. The MQ resource adapter file used by this Executor.

These three servers were created in the standard way – with the `server create` command of Liberty. The location of the Liberty product install directory was:

```
/shared/zWebsphere/Liberty/V8R55FP07
```

That's not shown on the picture. That location is referenced in the JCL start procedures, which we illustrate next.

What about the Angel process?

For this illustration we did not require an Angel process. We used JDBC Type 4 for access to DB2, and client mode connections to MQ. Those do not involve access to z/OS authorized services. Nothing else in this environment required the Angel.

If we used JDBC Type 2 it would imply RRS, and that requires the Angel. If we used the WOLA-based `batchManagerZos` command line utility, that too would require the Angel. But we are using JDBC T4 and the Java-based `batchManager` CLI, so no Angel is required.

1 As noted earlier, this could be shared between the servers. We chose to illustrate each server having its own artifacts because it's possible your environment would be across separate physical servers where sharing would not be done.

2 See other notes about how sharing was possible but not done for this illustration.

SAF commands and the JCL start procedures

In this section we will review some of the z/OS-related elements of the runtime.

JCL start procedures

The JCL start procedures for each were simply the sample JCL procs supplied with Liberty z/OS, copied to the `SYS1.PROCLIB` and renamed.

For example, the JSRDISP proc looked like this, with that highlighted in yellow the lines we updated:

```
//JSRDISP PROC PARMS='jsrdisp'
//*------------------------------------------------------------------
//* INSTDIR - the path to the WebSphere Liberty Profile install.
//*           This path is used to find the product code and is
//*           equivalent to the WLP_INSTALL_DIR environment variable
//*           in the Unix shell.
//* USERDIR - the path to the WebSphere Liberty Profile user area.
//*           This path is used to store shared and server specific
//*           configuration information and is equivalent to the
//*           WLP_USER_DIR environment variable in the Unix shell.
//*------------------------------------------------------------------
//   SET INSTDIR='/shared/zWebSphere/Liberty/V8R55FP07'
//   SET USERDIR='/shared/jsrhome/liberty'
//*------------------------------------------------------------------
//* Start the Liberty server
//*
//* WLPUDIR - PATH DD that points to the Liberty Profile's "user"
//*           directory. If the DD is not allocated, the user
//*           directory location defaults to the wlp/usr directory
//*           in the install tree.
//* STDOUT  - Destination for stdout (System.out)
//* STDERR  - Destination for stderr (System.err)
//* STDENV  - Initial Unix environment - read by the system.  The
//*           installation default and server specific server
//*           environment files will be merged into this environment
//*           before the JVM is launched.
//*------------------------------------------------------------------
//STEP1   EXEC PGM=BPXBATSL,REGION=0M,
//   PARM='PGM &INSTDIR./lib/native/zos/s390x/bbgzsrv &PARMS'
//WLPUDIR  DD PATH='&USERDIR.'
//STDOUT   DD SYSOUT=*
//STDERR   DD SYSOUT=*
//*STDENV   DD PATH='/etc/system.env',PATHOPTS=(ORDONLY)
//*STDOUT   DD PATH='&ROOT/std.out',
//*            PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//*            PATHMODE=SIRWXU
//*STDERR   DD PATH='&ROOT/std.err',
//*            PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//*            PATHMODE=SIRWXU
//* ================================================================= */
//* PROPRIETARY-STATEMENT:                                            */
//* Licensed Material - Property of IBM                               */
//*                                                                   */
//* (C) Copyright IBM Corp. 2011, 2012                                */
//* All Rights Reserved                                               */
//* US Government Users Restricted Rights - Use, duplication or       */
//* disclosure restricted by GSA ADP Schedule Contract with IBM Corp.*/
//* ================================================================= */
```

The `INSTDIR=` variables was updated to point to where Liberty z/OS 8.5.5.7 was installed, and the `USERDIR=` variable was updated to point to where the server configuration resided. The `PARMS=` value on the first line was updated with the server name, which then allows the start command to be simply: `S JSRDISP` for the dispatcher server.

The `JSREXEC1` server's JCL was similarly updated:

```
//JSREXEC1 PROC PARMS='jsrexec1'
//*------------------------------------------------------------------
//* INSTDIR - the path to the WebSphere Liberty Profile install.
//*           This path is used to find the product code and is
//*           equivalent to the WLP_INSTALL_DIR environment variable
//*           in the Unix shell.
//* USERDIR - the path to the WebSphere Liberty Profile user area.
//*           This path is used to store shared and server specific
//*           configuration information and is equivalent to the
//*           WLP_USER_DIR environment variable in the Unix shell.
//*------------------------------------------------------------------
//  SET INSTDIR='/shared/zWebSphere/Liberty/V8R55FP07'
//  SET USERDIR='/shared/jsrhome/liberty'
//*------------------------------------------------------------------
//* Start the Liberty server
//*
```

(The rest of the JCL removed to save space in this document)

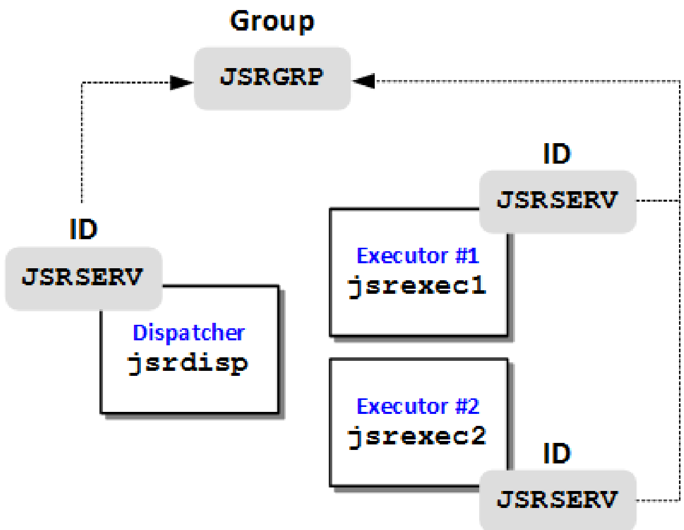
And the `JSREXEC2` server's JCL updated like this:

```
//JSREXEC2 PROC PARMS='jsrexec2'
//*------------------------------------------------------------------
//* INSTDIR - the path to the WebSphere Liberty Profile install.
//*           This path is used to find the product code and is
//*           equivalent to the WLP_INSTALL_DIR environment variable
//*           in the Unix shell.
//* USERDIR - the path to the WebSphere Liberty Profile user area.
//*           This path is used to store shared and server specific
//*           configuration information and is equivalent to the
//*           WLP_USER_DIR environment variable in the Unix shell.
//*------------------------------------------------------------------
//  SET INSTDIR='/shared/zWebSphere/Liberty/V8R55FP07'
//  SET USERDIR='/shared/jsrhome/liberty'
//*------------------------------------------------------------------
//* Start the Liberty server
//*
```

(The rest of the JCL removed to save space in this document)

Group and IDs

The group and started task IDs we used for this environment looked like this:



In other words, all three servers were assigned the same JSRSERV ID, and that ID was connected to the JSRGRP group.

Note: if you wanted separate IDs you could do that. We chose to keep things simple for this illustration and have one ID and one group.

The RACF commands we used to create that group and ID were:

```
ADDGROUP JSRGRP OMVS(AUTOGID) OWNER(SYS1)

ADDUSER JSRSERV DFLTGRP(JSRGRP) OMVS(AUTOUID HOME(/shared/jsrhome) -
  PROGRAM(/bin/sh)) NAME('LIBERTY SERVER')
```

Check with your security administrator to find out what is appropriate for your environment.

STARTED profiles

We created three JCL start procedures, one for each server. This was not required. We could have used the same JCL start procedure to start each server[3]. We chose to create separate JCL to more clearly and simply illustrate the relationship to the WLM STC classification rules for the Liberty servers[4]. We provide details starting on page 15.

The three JCL start procedures were named: `JSRDISP`, `JSREXEC1` and `JSREXEC2`. We provide details on those in the next section.

The RACF comamnds we used were:

```
RDEFINE STARTED JSRDISP.* UACC(NONE) -
  STDATA(USER(JSRSERV) GROUP(JSRGRP) -
  PRIVILEGED(NO) TRUSTED(NO) TRACE(YES))

RDEFINE STARTED JSREXEC1.* UACC(NONE)-
  STDATA(USER(JSRSERV) GROUP(JSRGRP) -
```

3 This would have worked because all three of our servers were under the same `WLP_USER_DIR`. The JCL `USERDIR=` variable points to that directory. For serves under the same directory the same JCL can be used. Different directories implies different JCL start procedures.

4 As you will see starting on page 15, we used the `TYPE=TN` for the STC classification rule, which relates to the `JOBNAME`. With a simple `S <proc>` the `JOBNAME` is the JCL proc name. There are other ways to achieve STC classification, but this was the most simple and straight-forward, so we chose that to illustrate in this document.

```
  PRIVILEGED(NO) TRUSTED(NO) TRACE(YES))
RDEFINE STARTED JSREXEC2.* UACC(NONE)-
  STDATA(USER(JSRSERV) GROUP(JSRGRP) -
  PRIVILEGED(NO) TRUSTED(NO) TRACE(YES))
SETROPTS RACLIST(STARTED) REFRESH
```

START commands

With the JCL start procedures in place (page 12) and the STARTED profiles created, the START command to start each server[5] was:

```
S JSRDISP
S JSREXEC1
S JSREXEC2
```

And the ID JSRSERV would be assigned to each.

WLM STC classification rules and service class goals

The objective was to have a different WLM service class goal assigned to each executor. This was to illustrate the principle of dispatching jobs to different servers based on a job parameter of jobClass= and have them run in the server based on the desired job priority.

Each server has its own JCL start procedure to make assigning a service class very simple when a started task is started with a simple S *<proc>*,PARMS='*server*'. In that case the JOBNAME defaults to the procedure name, and in WLM the TYPE=TN can be used to assign a service class based on JOBNAME.

Note: The WLM STC classification rule TYPE=PR may be used to assign based on procedure name. Or, you could start the server with S *<proc>*,JOBNAME=*<jobname>*,PARMS='*server*' and assign a JOBNAME different from the JCL start procedure name. We chose TYPE=TN and the procedure name becoming the job name to keep things simple.

The STC classification rules we coded looked like this:

```
          --------Qualifier--------               -------Class--------
Action    Type        Name      Start                Service     Report
                                          DEFAULTS:  OPS_LO      ________
 ____  1  TN          JSRDISP   ___                  JSR_HIGH    ________
 ____  1  TN          JSREXEC1  ___                  JSR_LOW     ________
 ____  1  TN          JSREXEC2  ___                  JSR_HIGH    ________
```

Notice that the JSREXEC1 server is assigned a service class of "low" while the JSREXEC2 server is assigned a service class of "high." That's the key point – the two executors will operate under different z/OS WLM goals[6].

The details behind each service class were:

Name	*Goal Type*	*Velocity*	*Importance*
JSR_HIGH	Velocity	80%	2
JSR_LOW	Velocity	10%	4

5 You may see other documentation show the start command as S *<proc>*,PARMS='*server*'. The PARMS= value passes the server name into the proc. The JCL start procedures we used included the PARMS= server name as part of the proc (see "JCL start procedures" on page 12). Therefore, our start commands were simply S <proc>.

6 The dispatcher is assigned the same "high" service class to keep things simple. In general the dispatcher classification should be equal to or higher than the classification of any of the executors behind it.

The specific details are not that important. The key is that they are different, and that we can influence where jobs are dispatched based on the message selector and the job parameter we pass in.

When we started the servers we saw our classifications in effect:

```
JOBNAME   StepName  ProcStep JobID    Owner    SrvClass
JSRDISP   JSRDISP   STEP1    STC00160 JSRSERV  JSR_HIGH
JSREXEC1  JSREXEC1  STEP1    STC00163 JSRSERV  JSR_LOW
JSREXEC2  JSREXEC2  STEP1    STC00164 JSRSERV  JSR_HIGH
```

The `JSRDISP` server received the `JSR_HIGH` service class, as did the `JSREXEC2` server. The `JSREXEC1` server received the `JSR_LOW` service class.

MQ Queue Manager and queue definition

In our environment the MQ queue manager was `MQS1` and the queue we defined was called `JSR.BATCH.QUEUE`. The key thing is the queue is marked *shareable*, since the dispatcher and all executors will operate against the same queue.

Work with your MQ administrator if you set this up in your environment.

Deployed batch applications

For this illustration we deployed the same sample applications – SleepyBatchlet and BonusPayout – into both executor servers. We did this by placing them in the `/dropins` directory of each server. For `jsrexec1` it looked like this:

```
/shared/jsrhome/liberty/servers/jsrexec1/dropins/
  Type   Permission  Owner      Filename
_ File    rwxrwxrwx  JSRSERV    BonusPayout-1.0.war
_ File    rwxrwxrwx  JSRSERV    SleepyBatchletSample-1.0.war
```

And for `jsrexec2` it looked like this:

```
/shared/jsrhome/liberty/servers/jsrexec2/dropins/
  Type   Permission  Owner      Filename
_ File    rwxrwxrwx  JSRSERV    BonusPayout-1.0.war
_ File    rwxrwxrwx  JSRSERV    SleepyBatchletSample-1.0.war
```

They are the exact same application WAR files in each server's /dropins.

As mentioned before, they could have been shared between the two executor servers since this environment had the servers under the same `WLP_USER_DIR` directory. We chose to illustrate them not shared to avoid any confusion for cases where sharing wasn't possible.

Dispatcher server.xml

In this section we will take a tour of the `server.xml` for the Dispatcher and describe what the key elements of the XML are doing. For a view of the `server.xml` in its entirety, see page 20.

Feature definitions

The `<featureManager>` elements define what features are specifically loaded to support the functions needed for the server. For the Dispatcher the features were:

```
<featureManager>
    <feature>servlet-3.1</feature>
    <feature>batch-1.0</feature>
    <feature>batchManagement-1.0</feature>
    <feature>appSecurity-2.0</feature>
    <feature>wmqJmsClient-2.0</feature>
</featureManager>
```

Where:

- `servlet-3.1` – this provides support for the batch-1.0 function.
- `batch-1.0` – this is what provides the essential JSR 352 support for the server.
- `batchManagement-1.0` – this provides the REST interface used by the `batchManager` command line utility, as well as the multi-JVM support for the dispatcher / executor model.
- `appSecurity-2.0` – the REST interface enabled with `batchManagement-1.0` has a security requirement, and to the `appSecurity-2.0` element is required to enable application security.
- `wmqJmsClient-2.0` – the Dispatcher function is going to use JMS to access the MQ queue to place the job submission messages. It needs this feature to do that.

Note: Liberty features often include other features, so in truth the following is all that's needed:

```
<featureManager>
  <feature>batchManagement-1.0</feature>
  <feature>appSecurity-2.0</feature>
  <feature>wmqJmsClient-2.0</feature>
</featureManager>
```

That's because `batchManagement-1.0` includes `batch-1.0`, and `batch-1.0` includes `servlet-3.1`. But Liberty is fairly tolerant of features specified when they're included with others. We're showing all the features to keep the function provided clear and obvious.

Basic security for Dispatcher function

The REST interface used by the `batchManager` command line utility has a few security requirements. For this illustration we chose to use the "basic" security configuration where everything is configured inside the `server.xml`.

Important: This method of configuring security works. It satisfies the minimum requirements. *But you would not likely use this mechanism in a production environment.* It is possible to use SAF or LDAP for the registry and role checking, and SAF or files for the key/trust stores. We chose to stay simple for this document to keep the focus on the main thing, which is the dispatching to the executor based on a job parameter.

The elements in the XML for basic security were:

```
<keyStore id="defaultKeyStore" password="Liberty"/>

<basicRegistry id="basic1" realm="jbatch">
  <user name="Fred" password="fredpwd" />
</basicRegistry>

<authorization-roles id="com.ibm.ws.batch">
  <security-role name="batchAdmin">
  <user name="Fred" />
  </security-role>
</authorization-roles>
```

Where:

- `<keyStore>` – this tells Liberty to use its own internally-generated and self-signed certificate for encryption on connections to this server. This certificate is not trusted since it is not signed by a well-known certificate authority (CA). But it is "good enough" for validation testing such as this.

- `<basicRegistry>` – this defines the userid and password for granting access to the server for submitting jobs. Here we are creating an ID of `Fred` and a password of `fredpwd`. We used this ID and password when submitting jobs.
- `<authorization-roles>` – this grants the ID access to the application role that allows it to submit jobs. The security role here is `"batchAdmin"`, and we are granting the `Fred` ID access to it.

Once again, this is called "basic" for a reason – it works, but it is definitely *not* recommended for anything that requires actual security. In those cases the key and trust store, the registry, and the application role would be in SAF or LDAP, and the `server.xml` would be configured accordingly[7].

Batch persistence and JDBC to the JSR job repository

For this environment we chose DB2 z/OS as the relational data store for the JSR 352 Job Repository function[8]. Further, we chose to use JDBC Type 4 as the connectivity mechanism. We chose that over Type 2 because Type 4 does not require the Angel process. Our objective was a relatively simple configuration so the key points we were trying to show would be clear.

The portion of the `server.xml` devoted to the batch persistence and JDBC Type 4 connection was the following:

```
<batchPersistence jobStoreRef="BatchDatabaseStore" />

<databaseStore id="BatchDatabaseStore"
  dataSourceRef="batchDB" schema="JBATCH" tablePrefix="" />

<jdbcDriver id="DB2T4" libraryRef="DB2T4LibRef" />

<library id="DB2T4LibRef">
  <fileset dir="/shared/db21010/jdbc/classes/"
  includes="db2jcc4.jar db2jcc_license_cisuz.jar sqlj4.zip" />
</library>

<dataSource id="batchDB" jndiName="jdbc/batch"
  type="javax.sql.XADataSource"
  jdbcDriverRef="DB2T4">
 <properties.db2.jcc
   serverName="wg31.washington.ibm.com"
   portNumber="9446"
   databaseName="WG31DB2"
   user="<userid>"
   password="<password>"
   driverType="4" />
</dataSource>
```

Where:

- `<batchPersistence>` – this enables the JSR 352 Job Repository to use a persistence data store[9]. It points to the `<databaseStore>` element.

7 See http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102110 for an illustration of using SAF for some of these functions.

8 The multi-JVM dispatcher/executor model requires shared access to the JobRepository. DB2 z/OS provided this relatively easily for this environment. DB2 LUW would have worked as well.

9 The *absence* of this element tells the Java Batch function to use the in-memory option. We can't use that for the dispatcher / executor illustration because that requires each server to share the same batch persistence tables.

- `<databaseStore>` – this points to the `dataSourceRef=` definition, and provides a few other important pieces of information, such as the table schema name.
- `<jdbcDriver>` – this points to the `<library>` element, which tells Liberty where the JDBC drivers are located.
- `<library>` – this provides information on where to get the JDBC drivers.
- `<dataSource>` – this provides the JNDI name the batch container uses to look up the datasource, and the `<properties>` element defines the DB2 z/OS system to talk to. Notice that here we are defining a Type 4 that communicates via DDF to a host and port and has a user and password alias for access into DB2.

  Notes: JDBC Type 2 is possible as well, but we chose not to use that because it would require the use of RRS and thus require the Angel. It was more detail than we wished to include for this illustration. It is possible to encode the password for the user. It is also possible to code an alias and remove the `user=` and `password=` from the properties.

This is really fairly standard Liberty definitions for accessing DB2 via JDBC Type 4.

The creation of the tables in DB2 to support the WebSphere Java Batch container is documented in the WP102544 Techdoc at `ibm.com/support/techdocs`.

Dispatcher function and JMS/MQ definitions

The Dispatcher uses JMS to communicate with the queuing function for job submission to the Executors. In our illustration we used IBM MQ as the queueing function[10].

The portion of the `server.xml` that defined the JMS resources was the following:

```
<batchJmsDispatcher
  connectionFactoryRef="batchConnectionFactory"
  queueRef="batchJobSubmissionQueue" />

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

<wmqJmsClient startupRetryCount="999"
  startupRetryInterval="1000ms"
  reconnectionRetryCount="10"
  reconnectionRetryInterval="5m">
</wmqJmsClient>

<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>

<jmsQueue id="batchJobSubmissionQueue"
  jndiName="jms/batch/jobSubmissionQueue">
  <properties.wmqJms baseQueueName="JSR.BATCH.QUEUE"
```

10 Any JMS-compliant queueing function is acceptable, provided it works with Liberty.

```
    priority="QDEF"
    baseQueueManagerName="MQS1">
  </properties.wmqJms>
</jmsQueue>
```

Where:

- `<batchJmsDispatcher>` – this defines the server as a Dispatcher and provides pointers to the JMS connection factory and the queue definition.
- `<variable>` – this provides a pointer to the location where the MQ JMS resource adapter is located. As we illustrated earlier, for simplicity we chose to locate this file in each server's configuration directory, even though we could have shared across all three. The `${server.config.dir}` variable resolves to the directory where this server's server.xml is located, and that's where we stored the MQ JMS RAR file[11].
- `<wmqJmsClient>` – this provides some settings the IBM Knowledge Center suggested helps insure a more robust JMS client. So we added them to our configuration.
- `<jmsConnectionFactory>` – this provides information on how to reach the MQ queue manager. In this illustration we're using MQ client mode.
- `<jmsQueue>` – this defines the queue to use when dispatching jobs.

The entire server.xml for the Dispatcher

```
<?xml version="1.0" encoding="UTF-8"?>
<server description="new server">

    <!-- Enable features -->
    <featureManager>
        <feature>servlet-3.1</feature>
        <feature>batch-1.0</feature>
        <feature>batchManagement-1.0</feature>
        <feature>appSecurity-2.0</feature>
        <feature>wmqJmsClient-2.0</feature>
    </featureManager>

    <keyStore id="defaultKeyStore" password="Liberty"/>

    <basicRegistry id="basic1" realm="jbatch">
      <user name="Fred" password="fredpwd" />
    </basicRegistry>

    <authorization-roles id="com.ibm.ws.batch">
      <security-role name="batchAdmin">
      <user name="Fred" />
      </security-role>
    </authorization-roles>

    <batchPersistence jobStoreRef="BatchDatabaseStore" />

    <databaseStore id="BatchDatabaseStore"
      dataSourceRef="batchDB" schema="JBATCH" tablePrefix="" />

    <jdbcDriver id="DB2T4" libraryRef="DB2T4LibRef" />
```

11 The location could also be a hard-coded pointer to any location in the file system. The only requirement is the server ID needs READ access to the file. Sharing across the three servers would have involved storing the file in the `/shared/resources` directory and using the built-in variable `${shared.resource.dir}`. Simple enough, but if the servers are not under the same `WLP_USER_DIR` then that variable would not work for sharing.

```
    <library id="DB2T4LibRef">
      <fileset dir="/shared/db21010/jdbc/classes/"
      includes="db2jcc4.jar db2jcc_license_cisuz.jar sqlj4.zip" />
    </library>

    <dataSource id="batchDB" jndiName="jdbc/batch"
      type="javax.sql.XADataSource"
      jdbcDriverRef="DB2T4">
     <properties.db2.jcc
       serverName="wg31.washington.ibm.com"
       portNumber="9446"
       databaseName="WG31DB2"
       user="<userid>"
       password="<password>"
       driverType="4" />
    </dataSource>

    <batchJmsDispatcher
      connectionFactoryRef="batchConnectionFactory"
      queueRef="batchJobSubmissionQueue" />

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

    <wmqJmsClient startupRetryCount="999"
      startupRetryInterval="1000ms"
      reconnectionRetryCount="10"
      reconnectionRetryInterval="5m">
    </wmqJmsClient>

    <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>

    <jmsQueue id="batchJobSubmissionQueue"
      jndiName="jms/batch/jobSubmissionQueue">
      <properties.wmqJms baseQueueName="JSR.BATCH.QUEUE"
        priority="QDEF"
        baseQueueManagerName="MQS1">
      </properties.wmqJms>
    </jmsQueue>

    <httpEndpoint id="defaultHttpEndpoint"
                  host="*"
                  httpPort="10000"
                  httpsPort="10001" />

</server>
```

Executor #1 server.xml

The executor's `server.xml` is very similar to the dispatcher's, but there are a few differences worth noting[12].

Feature definitions

These are the same as the dispatcher:

```
<featureManager>
    <feature>servlet-3.1</feature>
    <feature>batch-1.0</feature>
    <feature>batchManagement-1.0</feature>
    <feature>appSecurity-2.0</feature>
    <feature>wmqJmsClient-2.0</feature>
</featureManager>
```

The `appSecurity-2.0` feature is not needed for job submission through MQ. However, if you were to use the dispatcher function to fetch the job log, and you wanted to authorize based on the batch application role, you would need it. So we show it here for compleness.

Basic security for Executor function

When a job is dispatched to an executor, the identity of the submitter is passed as well. The executor function requires the ability to validate the identity against a registry. Keeping with the "basic" approach:

```
<basicRegistry id="basic1" realm="jbatch">
  <user name="Fred" password="fredpwd" />
</basicRegistry>
```

That's enough to give the Executor knowledge of the ID and verify it.

Batch persistence and JDBC to the JSR job repository

This is identical to the batch persistence definitions used by the Dispatcher (page 18). That makes some sense: the Dispatcher and Executor are sharing the same JobRepository information in the same instance of DB2. Therefore, the XML to access DB2 is the same.

```
<batchPersistence jobStoreRef="BatchDatabaseStore" />

<databaseStore id="BatchDatabaseStore"
 dataSourceRef="batchDB" schema="JBATCH" tablePrefix="" />

<jdbcDriver id="DB2T4" libraryRef="DB2T4LibRef" />

<library id="DB2T4LibRef">
 <fileset dir="/shared/db21010/jdbc/classes/"
 includes="db2jcc4.jar db2jcc_license_cisuz.jar sqlj4.zip" />
</library>

<dataSource id="batchDB" jndiName="jdbc/batch"
 type="javax.sql.XADataSource"
 jdbcDriverRef="DB2T4">
 <properties.db2.jcc
  serverName="wg31.washington.ibm.com"
  portNumber="9446"
  databaseName="WG31DB2"
  user="<userid>"
  password="<password>"
```

12 The difference between Executor #1's `server.xml` and Executor #2's is minimal. The key difference, as you'll see, is the message selector definition on the activation specification. This is what tells the Executor function which job submission messages to pick up.

```
  driverType="4" />
</dataSource>
```

JDBC for the BonusPayout sample application

Since the Executor has the BonusPayout application deployed, and that application's table is defined in the same DB2 as the batch persistence tables, we can simply add a *second* `<dataSource>` definition for the BonusPayout table:

```
<dataSource id="bonusDB" jndiName="jdbc/bonus"
 type="javax.sql.XADataSource"
 jdbcDriverRef="DB2T4">
 <properties.db2.jcc
  serverName="wg31.washington.ibm.com"
  portNumber="9446"
  databaseName="WG31DB2"
  user="<userid>"
  password="<password>"
  driverType="4" />
</dataSource>
```

That uses the same `<jdbcDriver>` and `<library>` definitions as were used by the batch persistence data source. The only difference is what's highlighted in yellow. The BonusPayout application will perform a JNDI lookup on `jdbc/bonus` to access its table.

Executor function and JMS/MQ definitions

The Executor JMS definitions are a little different from the Dispatcher's we reviewed starting on page 19. The Executor is not putting messages on the queue, it is listening for and getting messages. Its JMS definitions look like this:

```
<batchJmsExecutor activationSpecRef="batchActivationSpec"
  queueRef="batchJobSubmissionQueue"/>

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

<wmqJmsClient startupRetryCount="999"
  startupRetryInterval="1000ms"
  reconnectionRetryCount="10"
  reconnectionRetryInterval="5m">
</wmqJmsClient>

<jmsActivationSpec id="batchActivationSpec" >
  <properties.wmqJms
    destinationRef="batchJobSubmissionQueue"
    messageSelector="(com_ibm_ws_batch_applicationName = 'SleepyBatchletSample-1.0'
     OR com_ibm_ws_batch_applicationName = 'BonusPayout-1.0')
     AND (jobClass = 'A' OR jobClass IS NULL)"
    transportType="CLIENT"
    channel="SYSTEM.DEF.SVRCONN"
    destinationType="javax.jms.Queue"
    queueManager="MQS1"
    hostName="wg31.washington.ibm.com"
    port="1414">
  </properties.wmqJms>
</jmsActivationSpec>

<jmsQueue id="batchJobSubmissionQueue"
  jndiName="jms/batch/jobSubmissionQueue">
  <properties.wmqJms baseQueueName="JSR.BATCH.QUEUE"
```

```
    baseQueueManagerName="MQS1">
  </properties.wmqJms>
</jmsQueue>
```

Where:

- <batchJmsExecutor> – this defines the server as an Executor. It points to the JMS activation specification definition and the JMS queue definition in the XML.
- <variable> – this is the same pointer to the MQ JMS resource adapter as we saw before. As before, we have that RAR file located in each server's configuration directory. The `${server.config.dir}` variable resolves to that location.
- <wmqJmsClient> – this is the same set of values we saw for the Dispatcher. They were recommended by the IBM Knowledge Center to make the JMS client more robust.
- <jmsActivationSpec> – this is *different* from the Dispatcher. A JMS "activation spec" defines a function that listens on a queue and reacts based on the arrival of a message.

  Many of the properties of that definition are familiar: the transport type ("CLIENT"), the channel, the queue manager name, and the host and port of the queue manager.

  The property we wish to draw your attention to is the one higlighted in yellow. This is the message selector, and it defines what this Executor will pick up from the queue.

  In this case the message selector is defined to pick up any job submission message where the application name matches the strings for SleepyBatchlet or BonusPayout, and the `jobClass=` parameter is either A or not present.

  From the WLM classification definitions (page 15), we know that the `JSREXEC1` server will be assigned WLM service class `JSR_LOW`, which has a velocity of 10% and an importance of 4.

  If either SleepyBatchlet or BonusPayout is submitted with a job parameter of `jobClass=A`, then it will be run in this server. Therefore, `jobClass=A` implies a lower priority job classification.

  If either SleepyBatchlet or BonusPayout is submitted *without* a job parameter, then it will run in this server as well. That makes this lower-priority server a kind of "default" server for those two applications.

  The other executor is the "high-priority" server. It looks for `jobClass=B`.

- <jmsQueue> – this is the same as we saw for the Dispatcher. It defines the queue on the queue manager.

That is really the heart of this job classification mechanism. You specify a job parameter and you have a message selector coded in the server where the STC velocity goals will give the job the priority you wish it to have.

By the way, the job parameter `jobClass=A` is really just an arbitrary string. You could just as easily have a job parameter of `priority=low` or `velocity=high` or whatever. The important thing is having a properly coded message selector in the executor to "see" the parameter and pick up the job submission message.

The entire server.xml for Executor #1

```
<?xml version="1.0" encoding="UTF-8"?>
<server description="new server">

    <!-- Enable features -->
    <featureManager>
        <feature>servlet-3.1</feature>
        <feature>batch-1.0</feature>
        <feature>batchManagement-1.0</feature>
```

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

  <basicRegistry id="basic1" realm="jbatch">
    <user name="Fred" password="fredpwd" />
  </basicRegistry>

  <batchPersistence jobStoreRef="BatchDatabaseStore" />

  <databaseStore id="BatchDatabaseStore"
   dataSourceRef="batchDB" schema="JBATCH" tablePrefix="" />

  <jdbcDriver id="DB2T4" libraryRef="DB2T4LibRef" />
   <library id="DB2T4LibRef">
    <fileset dir="/shared/db21010/jdbc/classes/"
    includes="db2jcc4.jar db2jcc_license_cisuz.jar sqlj4.zip" />
   </library>

  <dataSource id="batchDB" jndiName="jdbc/batch"
   type="javax.sql.XADataSource"
   jdbcDriverRef="DB2T4">
   <properties.db2.jcc
    serverName="wg31.washington.ibm.com"
    portNumber="9446"
    databaseName="WG31DB2"
    user="<userid>"
    password="<password>"
    driverType="4" />
  </dataSource>

  <dataSource id="bonusDB" jndiName="jdbc/bonus"
   type="javax.sql.XADataSource"
   jdbcDriverRef="DB2T4">
   <properties.db2.jcc
    serverName="wg31.washington.ibm.com"
    portNumber="9446"
    databaseName="WG31DB2"
    user="SYSADM1"
    password="SYSADM1"
    driverType="4" />
  </dataSource>

  <batchJmsExecutor activationSpecRef="batchActivationSpec"
    queueRef="batchJobSubmissionQueue"/>

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

  <wmqJmsClient startupRetryCount="999"
    startupRetryInterval="1000ms"
    reconnectionRetryCount="10"
    reconnectionRetryInterval="5m">
  </wmqJmsClient>

  <jmsActivationSpec id="batchActivationSpec" >
    <properties.wmqJms
      destinationRef="batchJobSubmissionQueue"
      messageSelector="(com_ibm_ws_batch_applicationName = 'SleepyBatchletSample-1.0'
```

```
         OR com_ibm_ws_batch_applicationName = 'BonusPayout-1.0')
         AND (jobClass = 'A' OR jobClass IS NULL)"
        transportType="CLIENT"
        channel="SYSTEM.DEF.SVRCONN"
        destinationType="javax.jms.Queue"
        queueManager="MQS1"
        hostName="wg31.washington.ibm.com"
        port="1414">
      </properties.wmqJms>
    </jmsActivationSpec>

    <jmsQueue id="batchJobSubmissionQueue"
      jndiName="jms/batch/jobSubmissionQueue">
      <properties.wmqJms baseQueueName="JSR.BATCH.QUEUE"
         baseQueueManagerName="MQS1">
      </properties.wmqJms>
    </jmsQueue>

    <httpEndpoint id="defaultHttpEndpoint"
                  host="*"
                  httpPort="11000"
                  httpsPort="11001" />

</server>
```

Executor #2

The entire server.xml for Executor #2

This is very similar to the `server.xml` used by Executor #1, except the message selector is slightly different. We have that part highlighted in yellow.

```
<?xml version="1.0" encoding="UTF-8"?>
<server description="new server">

    <!-- Enable features -->
    <featureManager>
        <feature>servlet-3.1</feature>
        <feature>batch-1.0</feature>
        <feature>batchManagement-1.0</feature>
        <feature>appSecurity-2.0</feature>
        <feature>wmqJmsClient-2.0</feature>
    </featureManager>

    <basicRegistry id="basic1" realm="jbatch">
      <user name="Fred" password="fredpwd" />
    </basicRegistry>

    <batchPersistence jobStoreRef="BatchDatabaseStore" />

    <databaseStore id="BatchDatabaseStore"
     dataSourceRef="batchDB" schema="JBATCH" tablePrefix="" />

    <jdbcDriver id="DB2T4" libraryRef="DB2T4LibRef" />

    <library id="DB2T4LibRef">
     <fileset dir="/shared/db21010/jdbc/classes/"
     includes="db2jcc4.jar db2jcc_license_cisuz.jar sqlj4.zip" />
    </library>
```

```
<dataSource id="batchDB" jndiName="jdbc/batch"
 type="javax.sql.XADataSource"
 jdbcDriverRef="DB2T4">
 <properties.db2.jcc
  serverName="wg31.washington.ibm.com"
  portNumber="9446"
  databaseName="WG31DB2"
  user="SYSADM1"
  password="SYSADM1"
  driverType="4" />
</dataSource>

<dataSource id="bonusDB" jndiName="jdbc/bonus"
 type="javax.sql.XADataSource"
 jdbcDriverRef="DB2T4">
 <properties.db2.jcc
  serverName="wg31.washington.ibm.com"
  portNumber="9446"
  databaseName="WG31DB2"
  user="<userid>"
  password="<password>"
  driverType="4" />
</dataSource>

<batchJmsExecutor activationSpecRef="batchActivationSpec"
  queueRef="batchJobSubmissionQueue"/>

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

<wmqJmsClient startupRetryCount="999"
  startupRetryInterval="1000ms"
  reconnectionRetryCount="10"
  reconnectionRetryInterval="5m">
</wmqJmsClient>

<jmsActivationSpec id="batchActivationSpec" >
  <properties.wmqJms
    destinationRef="batchJobSubmissionQueue"
    messageSelector="(com_ibm_ws_batch_applicationName = 'SleepyBatchletSample-1.0'
     OR com_ibm_ws_batch_applicationName = 'BonusPayout-1.0')
     AND jobClass = 'B'"
    transportType="CLIENT"
    channel="SYSTEM.DEF.SVRCONN"
    destinationType="javax.jms.Queue"
    queueManager="MQS1"
    hostName="wg31.washington.ibm.com"
    port="1414">
  </properties.wmqJms>
</jmsActivationSpec>

<jmsQueue id="batchJobSubmissionQueue"
  jndiName="jms/batch/jobSubmissionQueue">
  <properties.wmqJms baseQueueName="JSR.BATCH.QUEUE"
     baseQueueManagerName="MQS1">
  </properties.wmqJms>
</jmsQueue>

<httpEndpoint id="defaultHttpEndpoint"
```

```
                    host="*"
                    httpPort="12000"
                    httpsPort="12001" />

</server>
```

The message selector here will pick up a job submission message for either SleepyBatchlet or BonusPayout, but *only* if the `jobClass=` parameter is B.

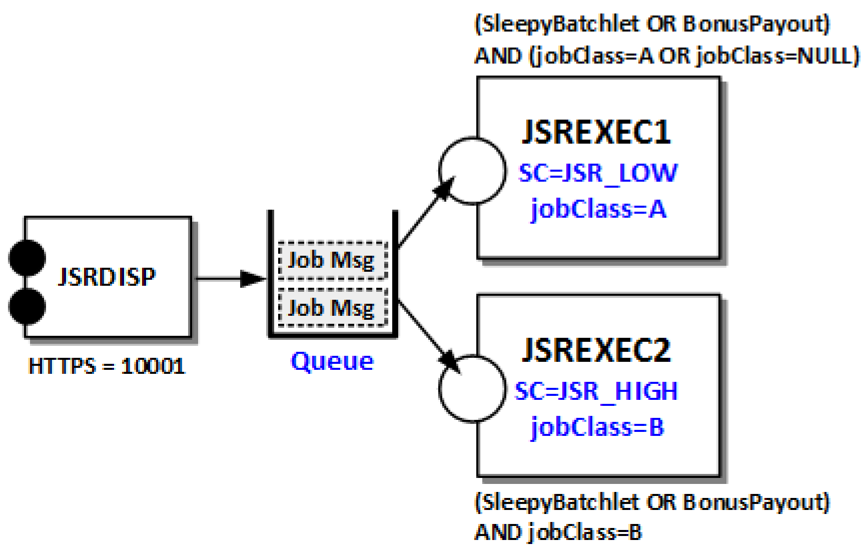
The other executor message selector allowed for a NULL jobClass, which is how that server was designated as the "default" for cases where no `jobClass=` is specified.

Job submission using batchManager CLI and evidence of expected message selection

The `batchManager` utility is a command line interface that provides a way to submit jobs through the Dispatcher function. We used that to verify our setup.

Basic requirements to use batchManager

Our server topology looked like this:



The `batchManager` command line utility needs access to a valid 64-bit Java as well as the access to the file in which the Liberty basic key/trust information is maintained. We did the following:

- We opened an SSH environment and logged in with the `JSRSERV` ID.
- We exported the `JAVA_HOME` variable to point to a valid 64-bit SDK:

  ```
  export JAVA_HOME=/shared/zWebSphere/Liberty/V8R55FP07/java/java_1.8_64/
  ```

- We exported JVM_ARGS to point to the key.jks file of the dispatcher server[13]:

  ```
  export JVM_ARGS="-Djavax.net.ssl.trustStore=/shared/jsrhome/liberty
                                  /servers/jsrdisp/resources/security/key.jks"
  ```

  Note: that second command is entered as *one line*. It was too long to fit on one line in this document so we split it across two lines.
  Because we logged on with the same ID used to create the dispatcher server (`JSRSERV`), we had READ down the file path and to that file.

13 You can avoid this by using `--trustSslCertificates` on the batchManager command. We opted to show you this method because it highlights how the client needs the SSL certificate.

Now we were ready to submit jobs and test our setup.

Test 1 – job submission with no jobClass= parameter

The first test was to submit the SleepyBatchlet job without a `jobClass=` parameter. The command we used was:

```
./batchManager submit --batchManager=localhost:10001
  --user=Fred --password=fredpwd --applicationName=SleepyBatchletSample-1.0
                                        --jobXMLName=sleepy-batchlet.xml --wait
```

That command was entered as *one line*.

That command has no --`jobParameter=jobClass=`*n* string as part of the command.

In our SSH session we saw:

```
[2015/11/06 17:59:27.598 -0500] CWWKY0101I: Job (NOT SET) with instance ID
62 has been submitted.

[2015/11/06 17:59:27.601 -0500] CWWKY0106I: JobInstance:{"jobName":"(NOT
SET)","instanceId":62,"appName":"SleepyBatchletSample-
1.0#SleepyBatchletSample-
1.0.war","submitter":"Fred","batchStatus":"STARTING","jobXMLName":"sleepy-
batchlet.xml","instanceState":"SUBMITTED"}
```

And then about 15 seconds later we saw it complete:

```
[2015/11/06 17:59:57.779 -0500] CWWKY0105I: Job (NOT SET) with instance ID
62 has finished. Batch status: COMPLETED. Exit status: COMPLETED

[2015/11/06 17:59:57.780 -0500] CWWKY0107I: JobExecution:
{"jobName":"sleepy-
batchlet","executionId":60,"instanceId":62,"batchStatus":"COMPLETED","exit
Status":"COMPLETED","createTime":"2015/11/06 22:59:31.011
+0000","endTime":"2015/11/06 22:59:46.618
+0000","lastUpdatedTime":"2015/11/06 22:59:46.618
+0000","startTime":"2015/11/06 22:59:31.439 +0000","jobParameters":
{},"restUrl":"SSL-ENDPOINT-
UNAVAILABLE","serverId":"localhost//shared/jsrhome/liberty/jsrexec1","logp
ath":"/shared/jsrhome/liberty/servers/jsrexec1/logs/joblogs/sleepy-
batchlet/2015-11-06/instance.62/execution.60/","stepExecutions":
[{"stepExecutionId":62,"stepName":"step1","batchStatus":"COMPLETED","exitS
tatus":"SleepyBatchlet:i=15;stopRequested=false","stepExecution":"https://
localhost:10001/ibm/api/batch/jobexecutions/60/stepexecutions/step1"}]}
```

That output indicated server `jsrexec1` is where it ran, which is what we expected since the submit command carried no `jobClass=` and thus the message selector for the the first executor server would be in effect.

SleepyBatchlet writes output to the `messages.log` file, and we saw evidence of it running in the `jsrexec1` server:

```
SleepyBatchlet: process: entry
SleepyBatchlet: process: sleep for: 15
SleepyBatchlet: process: [0] sleeping for a second...
SleepyBatchlet: process: [1] sleeping for a second...
SleepyBatchlet: process: [2] sleeping for a second...
SleepyBatchlet: process: [3] sleeping for a second...
SleepyBatchlet: process: [4] sleeping for a second...
SleepyBatchlet: process: [5] sleeping for a second...
SleepyBatchlet: process: [6] sleeping for a second...
SleepyBatchlet: process: [7] sleeping for a second...
```

```
SleepyBatchlet: process: [8] sleeping for a second...
SleepyBatchlet: process: [9] sleeping for a second...
SleepyBatchlet: process: [10] sleeping for a second...
SleepyBatchlet: process: [11] sleeping for a second...
SleepyBatchlet: process: [12] sleeping for a second...
SleepyBatchlet: process: [13] sleeping for a second...
SleepyBatchlet: process: [14] sleeping for a second...
SleepyBatchlet: process: exit. exitStatus:
SleepyBatchlet:i=15;stopRequested=false
```

The `jsrexec2` server messages.log had no such messages in it.

Conclusion: the job was submitted and it ran in the first executor as expected.

We next submitted the BonusPayout job with no `jobClass=` parameter. Again, we expected this to run in the `jsrexec1` server. The command we issued was:

```
./batchManager submit --batchManager=localhost:10001
           --user=Fred --password=fredpwd --applicationName=BonusPayout-1.0
     --jobXMLName=SimpleBonusPayoutJob.xml --jobParameter=dsJNDI=jdbc/bonus
                         --jobParameter=tableName=BONUS.ACCOUNT --wait
```

In our SSH session we saw:

```
[2015/11/06 18:13:32.280 -0500] CWWKY0101I: Job (NOT SET) with instance ID
63 has been submitted.

[2015/11/06 18:13:32.282 -0500] CWWKY0106I: JobInstance:{"jobName":"(NOT
SET)","instanceId":63,"appName":"BonusPayout-1.0#BonusPayout-
1.0.war","submitter":"Fred","batchStatus":"STARTING","jobXMLName":"SimpleB
onusPayoutJob.xml","instanceState":"SUBMITTED"}
```

And about 15 seconds later we saw:

```
[2015/11/06 18:14:02.540 -0500] CWWKY0105I: Job (NOT SET) with instance ID
63 has finished. Batch status: COMPLETED. Exit status: COMPLETED

[2015/11/06 18:14:02.542 -0500] CWWKY0107I: JobExecution:
{"jobName":"SimpleBonusPayoutJob","executionId":61,"instanceId":63,"batchS
tatus":"COMPLETED","exitStatus":"COMPLETED","createTime":"2015/11/06
23:13:32.461 +0000","endTime":"2015/11/06 23:13:34.554
+0000","lastUpdatedTime":"2015/11/06 23:13:34.554
+0000","startTime":"2015/11/06 23:13:32.514 +0000","jobParameters":
{"tableName":"BONUS.ACCOUNT","dsJNDI":"jdbc/bonus"},"restUrl":"SSL-
ENDPOINT-
UNAVAILABLE","serverId":"localhost//shared/jsrhome/liberty/jsrexec1","logp
ath":"/shared/jsrhome/liberty/servers/jsrexec1/logs/joblogs/SimpleBonusPay
outJob/2015-11-06/instance.63/execution.61/","stepExecutions":
[{"stepExecutionId":63,"stepName":"generate","batchStatus":"COMPLETED","ex
itStatus":"COMPLETED","stepExecution":"https://localhost:10001/ibm/api/bat
ch/jobexecutions/61/stepexecutions/generate"},
{"stepExecutionId":64,"stepName":"addBonus","batchStatus":"COMPLETED","exi
tStatus":"COMPLETED","stepExecution":"https://localhost:10001/ibm/api/batc
h/jobexecutions/61/stepexecutions/addBonus"}]}
```

That output indicated the `jsrexec1` server was used, as we expected.

The `messages.log` file does not show much evidence of BonusPayout running, but we can validate the job ran in that server by inspecting the job logs. Down the `/joblogs` directory path for the `jsrexec1` server we found the file, and inside we saw:

```
[11/6/15 23:13:32:513 GMT] com.ibm.ws.batch.JobLogger
```

```
==========================================================
Started invoking execution for a job
 JobInstance id = 63
 JobExecution id = 61
 Job Name = SimpleBonusPayoutJob
 Job Parameters = {tableName=BONUS.ACCOUNT, dsJNDI=jdbc/bonus}
==========================================================

[11/6/15 23:13:32:540 GMT] com.ibm.ws.batch.JobLogger
[11/6/15 23:13:32:579 GMT] com.ibm.ws.batch.JobLogger
==========================================================
For step name = generate
 New top-level step execution id = 63
==========================================================

[11/6/15 23:13:32:598 GMT] com.ibm.ws.batch.JobLogger
[11/6/15 23:13:32:644 GMT] com.ibm.ws.batch.JobLogger
[11/6/15 23:13:32:837 GMT] com.ibm.ws.batch.JobLogger
==========================================================
For step name = addBonus
 New top-level step execution id = 64
==========================================================

[11/6/15 23:13:32:892 GMT] com.ibm.ws.batch.JobLogger
[11/6/15 23:13:34:420 GMT] com.ibm.ws.batch.JobLogger
[11/6/15 23:13:34:570 GMT] com.ibm.ws.batch.JobLogger
[11/6/15 23:13:34:570 GMT] com.ibm.ws.batch.JobLogger
==========================================================
Completed invoking execution for a job
 JobInstance id = 63
 JobExecution id = 61
 Job Name = SimpleBonusPayoutJob
 Job Parameters = {tableName=BONUS.ACCOUNT, dsJNDI=jdbc/bonus}
 Job Batch Status = COMPLETED, Job Exit Status = COMPLETED
==========================================================
```

The JobInstance and JobExecution numbers[14] from the job log file matched what we saw in the SSH session output stream. Further proof the job ran in the first executor server as we expected.

Both SleepyBatchlet and BonusPayout ran in the `jsrexec1` server as we expected. The jobs ran there because we submitted the jobs with no `jobClass=` job parameter. The message selector for `jsrexec2` server requires a `jobClass=B` value, while the message selector for the `jsrexec1` server allows for `jobClass=A` or *no job class at all*.

Test 2 – job submission with jobClass=A parameter

The test here was to submit the same two jobs, but this time with:

`--jobParameter=jobClass=A`

added to the command string.

We expected both jobs to run in the `jsrexec1` server. The message selector match would be on:

```
messageSelector="(com_ibm_ws_batch_applicationName = 'SleepyBatchletSample-1.0'

    OR com_ibm_ws_batch_applicationName = 'BonusPayout-1.0')
```

14 The numbers were in the 60-range because we had run a lot of other tests prior to capturing output for this document. With a fresh set of JobRepository tables the first job submitted would have numbers of 1 and 1.

```
    AND (jobClass = 'A' OR jobClass IS NULL)"
```

from the `server.xml` for that server.

The command to submit SleepyBatchlet was:

```
./batchManager submit --batchManager=localhost:10001
  --user=Fred --password=fredpwd --applicationName=SleepyBatchletSample-1.0
          --jobXMLName=sleepy-batchlet.xml --jobParameter=jobClass=A --wait
```

Using the same validation test as before, we saw evidence this ran in the `jsrexec1` server.

The command to submit BonusPayout was:

```
./batchManager submit --batchManager=localhost:10001
           --user=Fred --password=fredpwd --applicationName=BonusPayout-1.0
     --jobXMLName=SimpleBonusPayoutJob.xml --jobParameter=dsJNDI=jdbc/bonus
    --jobParameter=tableName=BONUS.ACCOUNT --jobParameter=jobClass=A --wait
```

Using the same validation test as before, we saw evidence this ran in the `jsrexec1` server.

Both SleepyBatchlet and BonusPayout ran in the `jsrexec1` server as we expected. Both ran there because of the `jobClass=A` parameter.

Test 3 – job submission with jobClass=B parameter

This test was essentially the same as Test 2, except with `jobClass=B` on the job submission command. In this case we expect the message selector match to be from the second executor's `server.xml`:

```
messageSelector="(com_ibm_ws_batch_applicationName = 'SleepyBatchletSample-1.0'
 OR com_ibm_ws_batch_applicationName = 'BonusPayout-1.0')
 AND jobClass = 'B'"
```

SleepyBatchlet command:

```
./batchManager submit --batchManager=localhost:10001
  --user=Fred --password=fredpwd --applicationName=SleepyBatchletSample-1.0
          --jobXMLName=sleepy-batchlet.xml --jobParameter=jobClass=B --wait
```

BonusPayout command:

```
./batchManager submit --batchManager=localhost:10001
           --user=Fred --password=fredpwd --applicationName=BonusPayout-1.0
     --jobXMLName=SimpleBonusPayoutJob.xml --jobParameter=dsJNDI=jdbc/bonus
    --jobParameter=tableName=BONUS.ACCOUNT --jobParameter=jobClass=B --wait
```

Both ran in the `jsrexec2` server as expected.

Document change history

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

November 16, 2015	Initial Version
November 23, 2015	Updated with assigned WP number.

End of WP102600