WebSphere Application Server

The Complete Java Batch Blog Posts

This document can be found on the web at:
https://www.ibm.com/support/pages/node/6560420

Version Date: March 1, 2022
See "Document change history" on page 24 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`

Many thanks go to Kirsten Meren for helping me get started blogging and Don Bagwell for his constant encouragement and support.

Contents

Introduction

From mid-2018 through 2022 I wrote a (mostly) weekly blog talking about the application programming aspects of JSR-352, the open standard for Java Batch. These were all posted on the IBM Middleware User Community (IMWUC). This paper is a compilation of all of those blog posts into a single document.

Please remember these were written as blog posts and are, therefore, pretty informal.

Errors and such are all my fault. These posts represent my understanding of the specification as it existed as I was writing.

After the text of all the posts, you can find the source code for the samples referenced in the “How To” post series. These are best used from the github repository where they really live, but I included them here to be complete.

For fans of the LinkedIn posts that advertised each new blog post, I’ve included a list of all the songs I referenced at the very end of this document.

The podcast version of these posts remains up, at least for now. I’d like to thank Conor Walsh for the music separator used between the intro and the weekly content. You can find him here:

https://www.facebook.com/conorwalshmusic/
https://www.youtube.com/channel/UCDmwN1630K73E-jEMOS8azA

Batch Jobs in Java? YES! Modernizing Batch Applications with JSR-352

When we think about batch applications, we usually think about ancient COBOL programs running in the middle of the night during the mysterious ‘batch window’. But a batch application is really just a fixed sequence of process steps that occur in some specified order and start on some event. It might be a bunch of COBOL running at 3 AM, but it might also be Java invoking a rules engine to process records in a file that showed up on an FTP site.

A Batch process generally does a lot of things. Maybe it processes a lot of records, doing the same basic thing for every record. Maybe it works through a complex sequence of small steps, making decisions along the way about which step is next based on what it finds as it goes.

A batch application might be launched by an Enterprise Scheduler at a particular time or triggered by some event (a file arrives). But a batch application might be triggered from some transactional event occurring. Suppose a servlet is processing a fund transfer and there is a complex sequence of fraud detection steps that have to occur. That processing could occur as a batch application kicked off by the servlet – which may or may not wait for it to finish.

If you want to write a batch application in Java, a good place to start is with JSR-352. This open standard provides a framework for developing batch applications: multiple steps, flow control between the steps based on the results of the current step, checkpoint/restart capability, concurrent execution of different steps, even concurrent execution of multiple copies of the same step, and more.

Of course, you could do all of that yourself in your own Java code. But why? JSR-352 provides a standard way to do it that is portable across various implementations and platforms. And more importantly, you don’t have to maintain all that infrastructure code! Just write the parts to read from (and write to) your data sources and then focus on the business logic. The flow control and concurrency are all described in a standardized XML file. And the whole thing can be bundled into a .war file just like your web applications (and maybe with that servlet that starts it).

I’ve barely touched on all the cool things that are in JSR-352. There’s a lot more, as well as some interesting tips and techniques that we might get into in further blog posts. There’s lots of exciting and surprising things to be found in….batch!

Batchlets: All the Things You Need to Know!

Let’s talk about something of substance, but we’ll start off easy. Every exploration of a new technology usually somehow involves some bit of code that prints “hello, world!”. Writing a batchlet is the way to do that.

What is a batchlet? It is the simplest programming artifact available to a JSR-352 batch application. When invoked as part of a job, a batchlet gets control, does whatever it does, and completes. The principle method you implement to have a batchlet is the *process* method. It gets control with no parameters and returns a String. That String will be used as the exit status for the step (we’ll get into exit status values in a later post).

What can you do with a batchlet? Pretty much anything you want. The *process* method is just Java. Often a batchlet is used for file manipulation. Maybe you need to copy a file (or a bunch of files) from one place to another. Or you need to sort the contents of a file using a sort utility. In fact a batchlet is a good place to invoke a utility. The *process* method is really just a wrapper around a call to some other thing.

Batchlets get used a lot to manage flow control within the job. We’ll talk about flow with a job later, but at this point just recognize that there will be points in the execution of a job where you need to make decisions. Does a file exist already or not? Is there already a row in the account table for this account number? Write code here that will go look at whatever it is you need to know and return an appropriate exit status that the job flow can use to go to the appropriate next step.

A batchlet can be used for pretty much anything you want. Just remember that there is no transaction wrapped around the processing, so if your batchlet does multiple things that need to be committed together, you need to create and manage your own transaction. And if you find you’re doing the same thing in a loop inside your transaction…you should be using a chunk step. Generally speaking, if it isn’t obviously chunk processing, use a batchlet.

See you next time!

Whoa! Reining in your Java Batch Batchlet

Last time we talked about how great batchlets are for doing all sorts of things. But how do you stop one that gets out of control? Or that just happens to be in control when operations wants to stop the job?

This requires a little bit of thought from the developer. But before we get to that, let's take a moment to recall how stop processing works when a batchlet step is executing as the `stop` operation is issued. Stop processing will mark the job as stopping to try to prevent it from doing anything new. In a chunk step that means after each read/process cycle. But for a batchlet there isn't any point in processing where the batch container gets control to stop execution. The container needs help from the batchlet itself.

What will happen is the `stop` method of the batchlet will be driven by the container on a separate thread from the one executing the batchlet. This means the batchlet instance has code executing on two threads at once. It is up to the application code to find a way to communicate between those two and get the batchlet to stop.

Here's a high-level view:

```
boolean stopping=false;

process( ) {

while (!stopping) {
   … do batchlet stuff
}

stop( ) {
  stopping=true;
}
```

And that sort of structure works great if your batchlet is doing something in a loop. But if your batchlet has made a call to some other thing, maybe a database or some utility, then there isn't anything the `stop` method can probably do to stop it. The `stop` method has to just kind of shrug, look at the batch container apologetically, and return. Hopefully whatever the batchlet called will return and the *process* method will complete and then the job will stop.

An Introduction to the Step Context

Each step that executes in a Java Batch job has a context associated with it. From the context you can get information about the step and even set a few things into it. In this and the next few posts we'll take a closer look.

First, we need to understand how you get to the `StepContext`. Like many things in JSR-352, it is accessed via injection. The actual injection technology can vary depending on the implementation of JSR-352, but from an application perspective you probably don't care very much. Any batch artifact that runs as part of a step (Batchlet, Item Reader/Processor/Writer, various listeners) can get the `StepContext`. You'll get to it by putting this into your batch artifact:

```
@Inject
private StepContext stepContext;
```

That will result in the local variable `stepContext` being set to the `StepContext` object for this step. You can then use it to call any of the defined step context methods.

We'll take a look at a few of the simpler ones now. The step name and step execution id can be retrieved from getter methods on the step context. These aren't really much use to you, but if you are logging messages it might be helpful to include that information, especially if this piece of Java code gets used in several different places.

You can also get metric values for the step. The metrics are all related to a chunk step and include the number of items read, processed, etc. This is, again, mostly useful for logging or perhaps for some sort of chargeback processing. I suppose you might also use it in a custom checkpoint algorithm to help make a decision about when it is time to checkpoint.

For today we'll just look at one more use for the `StepContext`, accessing the step properties. Every artifact you can specify in the JSL for a job can have properties associated with it. But you can also associate properties with the step itself. These properties are those you might want to be able to access from any artifact that is part of the step, including listeners. Properties specified at the step level can be accessed from any artifact that is part of the step by injecting the `StepContext` as shown above and using the `getProperties` method.

One final word of caution…each step has its own, unique `StepContext` object, but a partitioned step gets a separate `StepContext` object for each partition.

StepContext Persistent and Transient User Data

The `StepContext` we discussed last time also provides access to transient and persistent user data. As you might expect, these user data fields are there to help the application communicate with itself across batch artifacts within a step. One, the transient data, is just for use as the step is running. The other, the persistent data, can be useful across restarts of a failed job.

Setting and getting transient data is easy. To set the data you use the `setTransientUserData` method on the `StepContext` and provide a reference to any Java object you like. This object instance will be available to any batch artifact that is part of the step. Therefore, any object you put in there can be accessed across the step. That could come in very handy for communication between the different bits of Java code that make up your step.

However…remember the caution from last time…for a partitioned step, each partition gets its own `StepContext`. Therefore, transient user data set in one partition is not available via the `StepContext` in another partition. This might be good. If an `ItemReader` sets data to communicate with a chunk listener, it doesn't want that data confused by an `ItemReader` processing data in a different partition. Each partition is working on its own.

Persistent User Data is also pretty cool. You can't just set any object in here though. Because the data will be persisted, it has to be a serializable object so it can be flattened for storage. For a chunk step the data is persisted at the end of each chunk along with checkpoint data for the chunk. That's pretty handy.

For a batchlet the data is persisted at the end of the step. Well, what good is that? Remember that if a failed job is restarted it is possible, depending on what is in the JSL, for a step that completed successfully to be re-executed anyway. The persistent user data for a batchlet could be used to remember what happened on a previous execution and influence what happens on a restart if you want to re-execute the batchlet rather than just accepting the results of the previous execution.

If you're confused by that, don't worry…restart processing is pretty complicated, and we'll talk about it here eventually.

Setting the step Exit Status

The exit status for a step is your chance to communicate between the running application code and the batch container that is using the JSL to determine the flow through the job. We'll talk more about options for flow control in another post, but today I wanted to look at setting exit status.

First of all, how do you set it? The step exit status is accessible through the `StepContext`. That means that from any artifact running as part of the step (reader, processor, batchlet, various listeners, etc) you can access the exit status.

What can you set the exit status to? Well, it is a string so you can set it to any string value you like. But remember, the most likely use of the exit status is to control flow within the JSL, so whatever value you set is likely to show up in the JSL. You can wildcard in the JSL with '*' and '?', but you still might want to put a little thought into establishing some sort of standard. An exit status of "I am so happy this step worked" might be fun, but looks a bit silly in the JSL to branch on successful execution of the step.

Which brings us to, what sort of status are you trying to convey with the exit status? The most obvious is whether what you are doing in the step worked or not. A basic successful/failure indicator. You might just use "0" and "1" for that, or maybe "SUCCESS" and "FAIL" is more clear. If you've got shades of success (or failure) you might want to go with the somewhat traditional numeric values: 0, 4, 8, 12.

But it is possible you might need to communicate something more complex. Perhaps the step is checking for the existence of a row in a table. Now you need to convey both that you were (or were not) successful in reading the table, but whether the row was found or not. You might encode that in a number as above, but it might be better to take advantage of having a string to communicate both things clearly. Perhaps "SUCCESSFUL:YES", "SUCCESSFUL:NO", or simply "FAILED" could convey all three possibilities.

Two last things to ponder. First of all, the `StepContext` allows you to both set AND get the exit status. Why would you want to get it? Because it might already be set! You might normally set the exit status from the `afterStep` method in the `StepListener`, but a chunk step might have had an error and already set the status from some other bit of application code running as part of the step. Do you want to override that error status or not? Being able to get the exit status allows you to check and make a decision.

And finally, remember that the exit status from the last step executed as part of the job is NOT the exit status from the job itself. The exit status from the job is an entirely separate thing and needs to be set using the `JobContext`. We'll talk about that more in a later post.

Basic Step Flow Control

There are two very simple rules for flow control through a job. The first rule is that the first step is always the first step. Whatever step is found first in the JSL file is the first step to be executed. Ok, technically the first thing in the JSL might be a 'split' in which case the split executes first and the first step inside all the flows inside the split are the first step, all at the same time. But you get the idea…the first thing found is the first thing to execute. No starting in the middle somewhere.

The second rule is that if you want the job to go somewhere, you have to tell it where to go. If a step completes without any direction being given about which step is next, the job is done. A JSR-352 Batch job will never just fall through the JSL file one step to the next in the order they appear. Every step has to indicate a next step in some way, or we're done.

There are some fancy options to do conditional flow of control through the job, but we'll save those for another day. The easy case is for a step to simply indicate what step should run next when this step finishes, regardless how things turn out in the step. You do that by including the 'next' keyword in the 'step' element. The value of 'next' has to be the ID of another step somewhere in the JSL. Every step, flow, split, or decision can have an id, and so the next attribute can indicate that element should execute after the current one.

The only place this gets really tricky is with a split/flow. Remember that you can group a set of steps together as a flow and then nest multiple flows within a split. All the flows within a split can execute concurrently. If your flow has multiple steps, you still need to indicate where each step should go when it finishes. Control will not flow sequentially down through the flow steps in the order they appear. When a step in a flow completes that has no indicated next step, the flow ends.

What happens when all the flows within a split end? The job ends, unless…the split element that wraps the flows indicates a 'next'. That's often a decision element.

One more rule…if you are executing steps within a flow, any next step indicated must be contained within the same flow. You can't jump across to other flows or out of the split entirely.

So go with the flow, and we'll see you next time…

Flow Control Using Step Exit Status

Time to pull together some things we've talked about earlier. Last time we covered how control flows from one step to another through the 'next' keyword as part of a step. Before that we talked about how an application can set the step Exit Status through the `StepContext`. Now we'll look at how the executing application can influence flow through a job using the Exit Status.

Before we start, there's one more thing we need to know about- Batch Status. While the Exit Status is a string value set by the application, the Batch Status (still a string) is one of a set of specification-defined values that indicate what's going on with the job. The values are: STARTING, STARTED, STOPPING, STOPPED, FAILED, COMPLETED, and ABANDONED. Obviously, a running job can only have a few of those – a running job's Batch Status will never be "STARTING". Batch Status is interesting because…here's the important part…if the application doesn't set an Exit Status, the Batch Status value is used.

Now that we've got that in our heads, there are four transition elements you can include in a step. All of them key off the Exit Status value for the step (which might be the Batch Status value, but you knew that). The transition elements are processed in the order they are found in the JSL, and you can wild-card the status string (so a catch-all value of '*' at the end might not be a bad idea).

The first element is 'next'. You would code it like this:

```
<next on=”0” to=”Step2” />
```

Which says that if the exit status for the step is "0" proceed to the step whose id value is Step2. You can transition to a step, a flow, or a split (using the appropriate id value for the one you want to transition to).

The other three transition elements are: fail, end, and stop. All allow you to specify an Exit Status to match against. When a match occurs for one of these, the Batch Status for the Job is set appropriately. For fail, the job has failed and is non-restartable. For end, the ends successfully (this is how you conditionally end a job, as opposed to just running out of steps to execute). And finally, for stop, the job stops executing, but is restartable. You can also set the id of a step to restart on, if you want. (We'll get to job restart processing eventually). All three allow you to set an Exit Status for the job itself (different from the Batch Status of the job).

```
<next on=”COMPLETED” to=”NextBlogEntry” />
```

Using the Job Context Transient Data

We've talked about the `StepContext` and about the persistent and transient user data associated with it. In this entry we'll have a look at the `JobContext` and the transient data that goes with it.

First of all, you can inject the `JobContext` into any batch artifact exactly like we injected the `StepContext` (using the `JobContext` class of course).

```
@Inject
private JobContext jobContext;
```

This results in the `jobContext` variable being set to the `JobContext` object for this job. Mostly. Remember that the `StepContext` was a single object for the Step, unless the step is a partitioned step in which case each partition got a separate instance of the `StepContext` object? Well, the `JobContext` works much the same way. There is a `JobContext` object for the main thread of the job, but there are also separate `JobContext` objects for separate threads of execution in the job.

This is really important because the `JobContext` isn't necessarily as 'global' to the job as you might hope. Each partition of a partitioned step and each flow of a split/flow will get a separate `JobContext` object. I bring this up first because it makes the second part of this entry somewhat less exciting than I'd like.

The `JobContext` has a bunch of methods on it, but the two of interest here are `getTransientUserData` and `setTransientUserData`. These methods allow you to get and set any object into the `JobContext` and it is there for the life of this job execution. That means that if the job fails and is restarted, the transient user data, being transient, is gone for the restart.

But it feels like this would be a great way to pass data between steps in the job. And it is, as long as you stay away from the parallel execution features of JSR-352. So you can have Step1 of a job set any object you like into the `JobContext` transient user data and have Step2 get that same object back out. Passing data between steps is a very common thing to need to do and being able to do it without writing it to a file or a database or something is really quite handy.

Just remember that if the getter and the setter aren't running on the same thread, you won't have the same `JobContext` object and the data won't make the trip.

Using the Job Properties from the JobContext

Every element of the JSL that defines a batch job can have properties that can be injected into that artifact. A listener can have properties that are injected into it, an `itemReader` can have properties injected into it, even a `partitionMapper` can have properties injected into it. But it turns out the job itself can have properties. How do you get to those?

The `JobContext` comes to our rescue. We talked about the `JobContext` last time and about how it can be injected into any batch artifact. We noted that separate threads of execution in the job get their own `JobContext` object instance. But no matter how many instances of the `JobContext` you have, they all have access to the job properties.

After injecting the `JobContext` you just call the `getProperties` method and a Properties object will be returned to you. You can use that Properties object like any other and fetch values that are set in the JSL as properties at the job level. Often those properties get values from job parameters that were supplied when the job was submitted. That makes this a handy way for any part of the job to get access to the job parameters.

There are a couple of notes in the `JobContext` interface about the properties that are worth discussing. The first points out that the Properties object you get back isn't necessarily some magic global Properties object you can access from anywhere. It might be that every time you get the Properties for the Job, you get a new, unique Properties object back.

Why is that important? It means that you shouldn't SET properties into that object and expect them to go anywhere. It is tempting to use that Properties object to set values as a way to pass information to other parts of the job. But that isn't safe because not all the users are necessarily getting the same Properties object. The vague wording in the specification means that while some implementations MIGHT give you back the same object and so setting properties into it might work, you can't count on that behavior across implementations and so you shouldn't depend on it.

The other note just points out that the implementation might 'tuck' other properties of its own into the `JobContext` properties so you shouldn't be surprised to find things in there that weren't in the JSL. The `JobContext` is a handy place for the JSR-352 implementation to hang things that are associated with the job. So don't panic if you find something in there that you didn't put there.

See you next time!

What Happened? Job Exit Status

In an earlier post we talked about how to set the Exit Status for a step and how that status can be used to control the flow of the job from one step to another based on how the current step completed.

But what about the job itself? We need an answer to the question: Did the job work? Whether you are running a single instance of a particular job, or if it is a recurring job run on some controlled schedule, you need to know if the job worked or not. You might want to restart the job if it failed in some way that a restart might fix. Knowing that might drive some automation (or an Enterprise Scheduler) to restart the job. Or someone in management might be interested in how an important job completed.

The result of a job is contained in the Exit Status for the job. Like the Exit Status for a step, this value is a String and can, therefore, be anything you like. It can just be a traditional numeric value (0, 4, 8, 12) or something readable (SUCCESS!) or some clever encoding that tells you something specific. Whatever it is, if there is some automation looking at the job result, you need to set that string to something it can handle.

The final Job Exit Status is extracted from the `JobContext`. If a batch application wants to set the Job Exit Status, it can use the setter method that is part of the context to set it. The Job Exit Status can be set over and over again by different parts of the batch application, if your application does that. The last value set is the final value for the job.

What if the application doesn't set the Job Exit Status? Then the final Batch Status value for the job will be used as the Exit Status. That means the Exit Status will be one of STOPPED, FAILED, or COMPLETED. Which might be all you need.

But if it isn't, you can have the application set the status using the `JobContext` but think about where you might get control that you would know enough to set it. Any individual step might not be able to know enough about what went on in the whole job to have an opinion about the whole job. Of course, if the final step can only be reached if all previously executed steps were successful….

Probably the best place for an application to set the final Job Exit Status is to have some code in the `afterJob` method of the `JobListener`. This is a job-level artifact that is the final bit of application code that will get control before the job ends.

Or just leave it alone and let the Batch Status value become the Exit Status.

What's That? An Introduction to Listeners

The JSR-352 specification includes a whole list of things called Listeners. We'll look at some of them more closely in upcoming posts, but before we get into details, what is a Listener?

A Listener is a batch artifact that can get control at various points in the processing of a batch job. The simplest Listener is the `JobListener`. It has methods that get control before and after the job. The other Listeners provide a way to implement code that gets control at other interesting points. If you find yourself wishing you could just run a little bit of code between when "this" happens and "that" happens, look for a listener.

All of the Listeners have a way to get control before and after whatever thing for which they are a Listener. Many of them have an additional method that gets control when an error occurs. For example, the `ItemReadLister` has an `onReadError` method that gets control if the `ItemReader` throws an exception.

To specify a Listener, just include it in a `<listener>` element inside the `<listen-ers>` element inside the `<step>` (or the `<job>` for the Job level Listener). What you don't have to specify is which Listener it is. The JSL doesn't care. Just give it the list of all the Listeners you want to get control and the batch container will figure out which class implements which Listener interface and do the right thing.

You can also specify more than one of the same Listener. For example, a given Step could have two `StepListeners` that will both get control. Be careful with this. If more than one Listener of the same type is specified there is no guarantee they will get control in the order they were specified (or any other order!).

To implement a Listener, just code up a Java class that implements the interface (and then specify it in the JSL). But…a Java class can implement multiple interfaces. Can one Java class actually implement multiple Listener interfaces? Sure. When you specify the class in the JSL the batch container will discover which Listener interfaces it implements and call the right methods whenever appropriate. If you look carefully at the Listener interfaces you'll see that the method names were carefully specified so they won't collide if one class implements more than one interface.

And that's the basics of Listeners. Next we'll get into how to use specific ones..

Outside-In, The Job Listener

The `JobListener` is the only one that can be specified in the `<listener>` element at the `<job>` level. Of course, as we noted last time, you can have more than one of them if you want.

The `JobListener` is very cool because it is the very first and the very last opportunity for you to get control for a job. The `beforeJob` method will get control before anything else happens in the job. The `afterJob` method will get control after everything else is done in the job. It is the very last chance you have to do something in the job.

What sort of things might you want to do? Well, in the `beforeJob` method you can do any sort of setup or initialization that is required for things later in the job. You might also be able to do some verification or validation to be sure you want to actually let the job run. Perhaps some resource needs to be available or it is pointless to continue. An exception thrown from the `beforeJob` method will stop the job cold.

The `afterJob` method is probably more interesting. Of course it can act as the mirror for the `beforeJob` method and tear down and clean up any sort of setup thing that got done at the start of the job. But it also serves as your last chance to set the job's Exit Status.

We've talked about this a bit in an earlier post. The `afterJob` method in the `JobListener` is the only batch artifact that gets control on the way out of the job and can set an overall Exit Status value for the entire job. If you plan to use it this way, you'll need to give some thought to how you will communicate the overall success of the job to the Listener. It doesn't receive any parameters, but you do know that the `afterJob` method will run on the same thread that has been acting as the main job thread all along. Of course partitions and splits might have caused some parts to run on other threads (and maybe in different processes entirely).

One last thing to ponder about the `afterJob` method: it always gets control! The spec says that even if some other bit of the job throws an unhandled exception, the `afterJob` method will still get control. That means that you can be sure that it will have an opportunity to set the Exit Status and do any required cleanup – if it can determine what it needs to do – it isn't passed the error that happened.

Happy Listening!

I hear (foot) steps! The Step Listener

We're working our way through the various Listeners defined in the Java Batch (JSR-352) specification. Last time we looked at the `JobListener` and this time we'll come inwards a little bit and have a look at the `StepListener`.

As you might expect, the `StepListener` gets control around a step. The Java class that implements the listener is defined in the JSL as part of the step. That means that you don't define one giant step listener class that gets control for every step in the job. Instead, for any step that needs a listener, you can define a separate implementation. If you want to.

Remember that in the JSL you are just specifying a step-level listener and the batch container determines what listener it is and when to give it control based on the interface(s) it implements. So a `StepListener` can do double-duty as some other Listener type if you want, by implementing multiple Listener interfaces.

Ok, so what methods do we have? Just the two: `beforeStep` and `afterStep`. These, obviously, get control before any other processing happens in the step and after it is all done. The other step-level Listeners we'll discuss all get control inside of these two methods.

The `beforeStep`, much like the `beforeJob` method on the `JobListener`, is your chance to do initialization or otherwise do any pre-actual-step processing you need to do. An exception thrown here will fail the job, so if something is wrong that you can detect before the step gets going you can kill it right here.

The `afterStep` method is there to let you clean up after the step. It gets control even if there is an unhandled exception thrown during the processing of the step. This means you can rely on it to do any clean up you need, even if things don't go well.

And, as with the `JobListener afterJob` method, this is your chance to set Exit Status. Remember that each step has a separate Exit Status value. And remember that the step Exit Status has nothing to do with the Job Exit Status. The step Exit Status might be used to determine which step executes next (if any). Using the `StepContext` to set an Exit Status value for the step from the `StepListener afterStep` method is pretty common. This is your chance, after everything is over and done, to decide how things turned out. But remember – nothing gets passed to the `afterStep` method. Any information it uses to make a decision has to come through the `StepContext` or some other communication mechanism between parts of the step processing.

The remaining listeners all relate to processing in a chunk step. We'll get to those after we've talked about chunk processing. But we have a few other things to get to first….

Group Your Steps into Flows

The JSR-352 specification includes a concurrency feature generally referred to as split/flows. We'll look at how this allows for concurrency and more details of a split next time. For now, let's just have a look at a flow.

What is a flow? It is basically just a group of execution elements wrapped in `<flow>` tags to merge them together. This allows you to reference the entire flow as a single thing. The id of a flow can make it the target of flow control from another step. This allows you to group a bunch of related stuff together and, at some point in job execution, just go do "all that stuff".

Don't be tempted to think of this is sort of a sub-routine that you can branch to whenever needed. The parts of a flow are still steps in the job and you can only execute a particular step once in the job. If you branch back to the same flow more than once the job will fail.

A flow can also have its own flow control (a 'next' tag in the 'flow' element) that indicates where in the JSL control should go once the flow is complete. There is no way to do conditional flow control based on the outcome of the entire flow with just JSL. We'll talk later about how a decision element can be used for that.

Unless the flow is contained within a split, if there is no 'next' tag on the 'flow' then the job ends when the flow is complete, just like it does for a regular step with no indication of what to do next.

The steps within a flow can, of course, have all the usual control options, including passing control to other steps. But those target steps HAVE to be within the flow. The flow is a grouping of execution elements and any branching around between them has to happen entirely within the flow. Attempts to branch out of the flow will result in the job failing.

Finally, I've been careful to say that a flow contains "execution elements" and not just "steps". That's because a flow can contain any executable element, including steps, other flows nested inside this one, or splits (containing other flows). You can make quite a complicated mess. If you do, be careful to understand the scoping because a step inside a flow inside a flow can't branch outside the inner flow. Got that?

All that said, a flow isn't all that exciting until we get to the split. That's up next.

Splitting Up Is Easy To Do…

Well, that's a bit dark. What we're talking about this time is the split feature of JSR-352. Splits give us the ability to run multiple steps concurrently within the same job. Generally we think of a job as a list of steps that run in some sequence. But there might be times where you have two steps that really don't depend on one another. Step A can run before or after Step B, but both have to run before Step C. Wouldn't it be cool if you could run Steps A and B concurrently?

To define a split, you first have to group the steps that need to run together into flows. In our example that would just be two flows, one with Step A and another with Step B. But the flows could easily be more complicated and consist of a group of steps, some conditional flows, maybe even a nested split within a flow.

Once you have the flows defined in the JSL, one after the other, you simply wrap them in a begin/end split element and there you go. The split element will have an id and be the target of some flow control from elsewhere in the job. When execution reaches the split, the main job thread will pause, and work will be spun off to other threads to try to run the flows within the split concurrently. I say 'try' because of course there is no guarantee the system will actually be able to run them concurrently, but it might.

We've talked about some of the issues of multi-threading before, but let's revisit them here. First of all, each of these threads gets a separate instance of the `JobContext`. This gives you access to the Job Properties that are common across the job. However, remember that each thread has its own Transient User Data. You can use that to share data across steps within a flow, but not across the flows. Also remember that branches between steps in a flow have to stay within the flow.

Another consequence of having a separate `JobContext` is the inability to set the Job's Exit Status. The Exit Status for the job is controlled from the Job Context on the main job thread. Don't bother setting it within a flow in a split. Similarly having flow control elements that fail or stop the job are not advisable within flows in a split.

When all the flows within a split are finished, execution proceeds to the job element identified on the next attribute of the split. You might want that to be a `Decision` element. We'll talk about those next time.

Decisions, Decisions....Decider! (Part 1)

A `Decision` element in the JSL is used to specify a `Decider` batch artifact which you can use to make decisions about what to do next.

A `Decision` element is kind of like a special type of step in the JSL. You can reference it as the next element from a step, a flow, or a split. It cannot be the first step in a job. The purpose of a `Decision` element is to give you a special point in the processing of a job where you can make decisions about what to do next. You might decide to branch this way instead of that, or to end the job (for good or bad) right here, or maybe something has gone horribly wrong and the job just needs to stop so it can be restarted after some issue is resolved.

Well, that's all fine, but can't you do all that with a normal step? Kind of.. A simple batchlet step will get control and can set a Step Exit Status that can be used with flow-control elements in the step to decide where to go next or to end the job.

A `Decision` has a few extras that make it special. The first special thing is that the implementation (a `Decider`) gets control in its `decide` method with an array of `StepExecution` objects as a parameter. A batchlet gets nothing (except injected properties).

A `Decider` also gets to return a string. A batchlet returns nothing. Why does it return a string? Because a `Decider` gets to set the Exit Status for the whole Job. That makes the Decider an interesting option as the final step in the job.

We still need to talk about what that `StepExecution` object is that the Decider gets as a parameter and why that would be an array…. Something for the next post…see you then!

Decisions, Decisions…Decider! (Part 2)

Last time we were talking about making decisions in the flow of a job using a `Decision` element in the JSL which executes an implementation of the `Decider` interface. That `Decider` gets passed as a parameter an array of `StepExecution` objects. This time we'll take a peek inside the `StepExecution` object and ponder why you might get an array of them.

A `StepExecution` is a little like a `StepContext`. It has information about a step like the step name, when it started and ended, and what the Batch and Exit Status values were. Well, that's fine…which step's `StepExecution` object gets passed to the `Decider`?

That's where it gets tricky. In general, the `Decider` gets passed the `StepExecution` object from the step that preceded it. If you have a simple step that runs a batchlet and then flows to a `Decider`, the `Decider` will get the `StepExecution` object from that batchlet step. In essence, here's how this step ended, now decide what to do.

The parameter is an array because more than one thing might flow to a `Decider` at the same time. Suppose you have a `Split` with several `Flows`. The next attribute on the `Split` can point to a `Decision` element. That `Decider` will get control with the array containing `StepExecution` objects for the final step of each `Flow` within the `Split`. That allows you to decide what to do next based on how all the `Flows` turned out.

The `StepExecution` also contains the `Persistent User Data` from the `Step`. Remember that persistent data has to be serializable. This allows the previous step (or the final step in each flow of a split) to pass an entire object to the `Decider`. Which can be a huge help as the `Decider` tries to figure out what to do.

Let's wrap this up then…a `Decider` can get control after previous things have executed in the job, whether that previous thing is a simple step, a flow, or several flows within a split. The `Decider` gets information about all the Steps that ran just prior to it, even if there are more than one of them across several concurrent threads. That information includes the `Persistent User Data` from those Steps. The `Decider` can set an Exit Status that can be used for flow control within the job AND also sets the Exit Status for the job itself (unless something later on overrides it).

Pretty powerful stuff.

Let’s Set Some Parameters…

In the next few entries we’re going to have a look at JSL and how you can substitute values into the JSL. There are a lot of different reasons to do that and some interesting possibilities. But before we get into all the details, let’s have a look at where substitution values can come from.

Our first source of substitution values comes from outside the job. They come from whoever submitted the job. Different implementations might provide some alternate ways to submit jobs, but the only way provided in the specification is through the ‘start’ method on the `JobOperator` interface. We’ll get to `JobOperator` eventually, but for now just know that the ‘start’ method allows you to provide a `Properties` object containing job parameters.

These parameters are key/value pairs (as you’d expect in a `Properties` object), where the key and values are both `Strings`. Be careful with this. The key and value for a `Properties` object are supposed to be `Strings`. However, since `Properties` inherits from `Hashtable` you can use the ‘`put`’ method to add things in there. Don’t do that. It is discouraged in general for `Properties` objects, but JSR-352 expects all Job Parameters found in the `Properties` object to be `Strings` and any other type won’t resolve when you try to use it.

That said, you can provide Job Parameters to direct flow control within the job or to pass values into executable parts of the job (i.e. to a batchlet or other artifact). We’ll see the actual syntax to process Job Parameters in a later post. For now, just realize that basically any value used in the JSL can include strings that are resolved from Job Parameters. That includes not just `Properties` injected into batch artifacts but other values like package and class names used in the ‘ref’ attribute of an executable element (like a batchlet). I’m not entirely sure why you would want to do that, but you can.

We haven’t gotten to job restart processing yet either, but the specification also allows for different Job Parameter values to be specified on the restart of a failed job than were used in the initial start. That could be handy if there are differences in how things need to work in a job doing restart processing. Might also be a handy way to let the job itself KNOW it is being restarted…

Well, that’s enough of an introduction to Job Parameters and a lot of hints about things to come…stay tuned!

Setting and Using Job Properties

Every programming artifact in the Java Batch environment allows you to specify properties in the JSL that will be injected into that artifact. You can have JSL properties that get injected into your batchlet, your `ItemReader`, your listener (of whatever type), etc. Job Properties are unique in that they don't get directly injected anywhere.

A Job Property exists to be used as a variable whose value is accessible across the entire job. You can use it to substitute a value into another property that is injected into an artifact.

For example, suppose you have a file that is used as input by several steps in the job. You don't want to hard-code the path to the file in the application. You want to inject it into the application from the JSL. But you also don't want to put the same value as a property in several different places in the same JSL. The solution is to set the path as a job property and then use that value to set the properties that are injected into the different artifacts in the job.

You can also set a job property from a job parameter. This makes a nice way to introduce the value of the parameter into the job in a manner that makes it available to the whole job. The ability to set a default value in the JSL also means you don't have to write code, possibly in several places, to know the default. The application code can assume a value always gets set.

Before we go, I thought I'd also remind you that all the Job Properties are available programmatically from within the job by accessing the `JobContext` object. It is probably cleaner to have the properties used to substitute into batch artifact properties that are then injected which separates the application code from knowing too much about the job level properties. But if the need arises, it is still possible to get them.

That's it for this time. You might be wondering about all these parameters and properties and default values and how you actually code these in the JSL. Good question. We'll take that on next time…

Substitution Syntax

So far in this blog we’ve stayed pretty high level and not gotten down into the details, focusing more on concepts and best practices (or at least I’ve tried to do that). But in this post we’re going to dig into the specific syntax of doing string substitution in Job Specification Language XML documents (JSL). Why? Because it looks pretty scary when you see it the first time and it really isn’t that bad…mostly.

The first thing to know is that you can substitute into pretty much any string value in the JSL. Suppose you have a property that you set like this:

```
<property name=”A” value=”ValueA” />
```

You can set either of those strings to instead include strings that are substituted into that string. For example:

```
<property name=”A” value=”Value#{jobProperties[‘someLetter’]}”
/>
```

That icky mess says that whatever value the Job Property called ‘someLetter’ has should be substituted into the string. So if someLetter is set to ‘X’ then the value of property A will be ‘ValueX’. On to the syntax.

It isn’t too bad if you take it in pieces. A substitution is contained inside the ‘#{‘ and ‘}’ delimiters. The ‘#’ indicates we’re doing substitution and the open/close curly braces show the start and end. Inside the braces you tell it where to get the substitution value from. You have four choices. You can get the value of a Job Parameter or a Job Property (see the previous two blog posts). You can also get the value of a System Property (specified with a ‘-D’ when the JVM was started). The final option involves partitions, which we’ll get to later on.

So take a look at our example above once more…. We see the ‘#{‘ that starts the substitution and the ‘} that ends it. And inside it the syntax says that we want a Job Property value. The name of the property to use is inside the square brackets and single quotes (someLetter).

Next time we’ll have a look at how to set up a default value. To close out here I’ll give you a few more samples. See if you can figure out how they resolve (answers next time). Assume a Job Parameter of “XX” is passed in with a value of “ZZ”.

```
<job id=”job1” >
    <properties>
        <property name=”ABC” value=”#{jobParameters[‘XX’]}” />
        <property name=”DEF” value=”ABC#{jobProperties[‘ABC’]}”
/>
    </properties>
```

```
<step id=”step1”>
    <properties>
        <property name=”GHI” value=”XYZ#{jobProper-
ties[‘ABC’]}XYZ#{jobProperties[‘DEF’]}XYZ”/>
    </properties>
</step>
</job>
```

Substitution Syntax Continued: Defaults

First the answers to last week's property mess. Starting at the top, the value of property ABC resolves to the value of the passed in parameter XX which we said had a value of ZZ. So ABC's value is "ZZ". Next, the value of DEF is the string "ABC" concatenated with the value of the Job Property ABC. Well, that was "ZZ" so the value of the DEF property is going to be ABCZZ.

Moving into the step, things got really messy. The value of the GHI property is the string "XYZ" concatenated with the value of the ABC property (ZZ) concatenated with "XYZ" again and then the value of the DEF property (ABCZZ) concatenated with "XYZ" (again). So the net for GHI is the string "XYZZZXYZABCZZXYZ".

But what would happen if the job was started without a value being supplied for the XX job parameter? As coded, the value of ABC would be an empty string. Then DEF would be shortened to just "ABC" and the GHI property would be set to "XYZXYZA-BCXYZ" (basically missing the ZZ value). Could we set up a default value to be used instead of an empty string? Sure!

The default is indicated by following the closing curly brace with a question mark and a colon, '?:' and terminating the value with a semi-colon ';'. Here's a simple example that sets the value of property A to "Z" if the myParameter Job Parameter isn't set:

```
<property name="A" value="Value#{jobParameter['myParame-
ter']}?:Z;" />
```

So all that will resolve to "ValueZ" if the Job Parameter isn't set.

Alright, so in the puzzle from last week we'll change the line that sets up the Job Property ABC to look like this:

```
    <property name="ABC" value="#{jobParameters['XX']}?:MM;" />
```

What does that do to our results if the XX Job Parameter isn't set? Come back next week to check your answer…

Substitution Syntax – More Realistic Examples

First, the answers to last week's problem:

ABC is MM because the Job Parameter XX isn't set.
DEF is therefore ABCMM.
GHI is thus XYZMMXYZABCMMXYZ.

I hope.

This is all very clever and hopefully you're more comfortable with the syntax by now, but what good is all this? Suppose you have a job with several steps that all produce output and you want that output to go into a common directory. You've got a directory in mind where you want it to go, but you'd like to be able to override that in some situations.

As an added twist, you have two different versions of the application and you'd like to use a single JSL to run both with the version controllable by the submitter of the job. The java class names are 'Scan' and 'Reduce' and they are either in the com.ibm.xapp1 or com.ibm.xapp2 package depending on which version you want.

I've kept the names of things terse to try to make this readable in a reasonable width.

```
<job id="job1" >
    <properties>
        <property name="OutDir" value="#{jobParameters['Out-
Dir']}?:/u/xapp/output/;" />
        <property name="Version" value="#{jobParameters['Ver-
sion']}?:1;"/>
    </properties>

    <step id="step1" next="step2">
        <properties>
            <property name="OutFile" value="#{jobProperties['Out-
Dir']}scanout.txt"/>
        </properties>
        <batchlet ref="com.ibm.xapp#{jobProperites['Version']}.Scan"/>
    </step>

    <step id="step2" >
        <properties>
            <property name="OutFile" value="#{jobProperties['Out-
Dir']}reduceout.txt"/>
        </properties>
        <batchlet ref="com.ibm.xapp#{jobProperites['Version']}.Re-
duce"/>
    </step>
```

Some observations…

- The Job Property names are the same as the Job Parameter names. This isn't required but makes it a lot easier to keep track of things

- The use of the Job Properties later in the job don't have a default because they don't need one. The default is set in the Job Property. We could have skipped the Property entirely, but having the default in one place at the top makes it easier to see what's going on (and avoid accidently having different defaults in different places, unless that's what you want!)

- We used substitution right there in the package name of the batchlet, right in the middle. That's actually ok, and pretty cool.

- Output goes into a common directory, but two different file names. Remember that the value of OutFile is just injected into the batchlet. It is up to the application to actually use the property as the location for whatever it writes.

- If you just run this job as-is twice, the output from the second run will use the same files as the first one unless the application code notices (or that's what you want).

Hopefully all this has helped you understand how to use substitution in your JSL. Enjoy!

One Chunk at a Time

Most descriptions of JSR-352 begin with a talk about the chunk step type, so I guess it is about time we got around to discussing it. We'll get into details in the following posts, but we'll start with an overview here.

The other step type, batchlet, is pretty simple. Your application code gets control, does whatever it does, and the step ends. A chunk step, on the other hand, involves multiple bits of application code, a loop, and transactions managed by the batch container.

The chunk step matches a pretty typical batch programming model that reads records, does some processing with each record, and then writes some results somewhere. And the JSR-352 model breaks the application code down into those three main parts: reader, processor, and writer.

The batch container loops calling those application artifacts over and over. As it does that, it creates a transaction around the processing done by the reader/processor/writer and commits that transaction at checkpoints. How often it checkpoints is configurable a few different ways that we'll look at later.

When the transaction commits, it will commit any updates done by the application (probably by the writer) that are transactional. But it also gets checkpoint data from the reader and writer (like what record number we just read) and commits that into the Job Repository where the batch container is keeping track of the progress of the job.

The checkpoint data allows a job running a chunk step to be restarted and be able to pick up again at the last checkpoint. That's because on a restart of a chunk step the reader will be provided with the last committed checkpoint data so it can just go read the next record after that, rather than start all over again from the beginning.

In upcoming posts we'll take a closer look at the specifics of the reader, processor, and writer, as well as different ways to specify the checkpoint interval, how errors are handled, listeners, and more!

Check(point) back regularly to find out more about it… (gak – can't believe I wrote that).

Writing a Chunk-Step Reader

You'll write an implementation of an `ItemReader` to fetch data to process in the `ItemProcessor` part of the chunk step. There are only four methods to implement. Let's take a look…

We'll start in an odd place, with the `checkpointInfo` method. This method will get called at every checkpoint to provide information about where the reader is processing its way through the source of the data. It has to return a serializable object because the data will get flattened and persisted in the Job Repository along with other information about the progress of the step (or partition when we get to those). What information should be in the `checkpointInfo`? Anything the reader needs to pick up where it left off. That might be a record number or an account number or an offset into something. Whatever you might need to know where you were. And it might be that nothing works because the data can change underneath you and you just have to start over.

Whatever `checkpointInfo` you provide will be given to the `open` method when it is called. When a job is run the first time the parameter will be null because there is no data from a prior execution, but if you provide `checkpointInfo`, your `open` method should be prepared to use it. That might mean positioning a cursor or reading your way into a file to a particular record or whatever is appropriate for your data source. The `open` method runs inside a transaction that is separate from the chunk transaction so anything you do in this method will commit before chunk processing starts.

Of course, the `open` method is matched by a `close` which is your opportunity to close files or database connections or whatever you did in open processing to establish your data source. Close processing also happens inside a transaction that is separate from the chunk processing transactions.

Finally, the `readItem` method is where you….read an item to process! This method gets called every time through the chunk processing loop and is expected to return an item to be processed. Be sure to advance however you are keeping track of where you are in the data so your checkpoint information will be current.

Remember that `readItem` is called repeatedly inside a transaction that will commit at the end of the chunk. If your cursor that is keeping track of where you are closes on a commit you will lose your place. You can get around that by configuring the data source you are using as non-transactional so the commit won't affect your cursor.

That's it for this time. Next post we'll look at the `ItemProcessor`

Processing Data with an ItemProcessor

The first thing to remember about the `ItemProcessor` is that it is optional. You can define a chunk step with just a reader and a writer. That makes sense because you may have cases where you are just reading data from here and writing it over there and there's no real "processing" to do on the data. So it is ok to just leave this out.

If you have a processor, you should carefully consider what data will be provided to it. Obviously it is the record that the `ItemReader` read in his `readItem` method. But what is that, specifically? You might just think that it is the record you read. That might be data object built from a row in a database or it might be a String (maybe JSON). But remember that this is how you communicate between the reader and processor. Creating an object to wrap the read data allows you to include other attributes (flags, etc) that might be handy. Not sure what that might be? Couldn't hurt to have a wrapper class anyway so you can have a place to add things later when you realize you need something.

There's also a temptation to write a 'generic' reader that reads from some data source you use a lot and provides its input to a bunch of different processors. That's pretty handy and allows you to build new batch jobs by putting together existing readers and processors. But remember that making these artifacts generically usable for different purposes will sometimes mean code executes in one use that is only needed in some other use. Might not seem like a big deal, but remember that the reader and processor will execute over and over for every record you process. A few extra bits of code, executed a few million times unnecessarily, can substantially add to the elapsed execution time for the step. So remember to think about cost when you are trying to write reusable batch artifacts.

What are we talking about? Oh right, the `ItemProcessor`. It just has one method, `processItem`. And it does whatever processing you need it to do. There's not much to say about that. You know what you need to do to the data and this is where you do it. Call a rules engine, do some data transformation, ask Watson for some analysis, do fraud detection, check inventory…whatever. This is where the action is.

When you are done, if you have something to write you return an object that will get passed to the `ItemWriter` to write. We'll see that the writer is getting all the objects to write at once at the next checkpoint so these are piling up in memory until the checkpoint. Don't get crazy with extra data in the write-object if your checkpoints could involve a lot of processed data.

We'll look at the writer next…

Writing results with an ItemWriter

The `ItemWriter` has a lot in common with the `ItemReader`. It has an `open` and a `close` method which get control at the beginning and end of the step (and in some retry cases we'll get to later). It also has a `checkpointinfo` method that gets called at every checkpoint.

As with the `ItemReader`, the writer can return any serializable object as checkpoint data and that data will be provided to the `open` method when the writer needs to pick up where it left off after a failure. If the writer is writing to a transactional resource, then the checkpoint data will be committed to the Job Repository along with whatever application data was written. However, if the target of the writer is non-transactional (like a flat file) then updates to the file aren't coordinated with the commit of the checkpoint data. If that's the case, on a restart you might have to compare the checkpoint data with what is actually in the file and possibly remove some updates to get things back in sync.

Of course the main method of the `ItemWriter` is `writeItems`( ). The `ItemReader's readItem` method is called to read one item, but `writeItems` is plural, emphasizing that it writes more than one. The signature for `writeItems` says that it gets a `List` of objects to write. As the chunk has progressed with `readItem` and `processItem` being called again and again, the results returned from `processItem` (or `readItem` if there's no `ItemProcessor`) have been piling up in memory. When a checkpoint is reached, those result items are placed into a `List` and passed to the `writeItems` method.

Why do it this way? The specification could have said to just call the `ItemWriter` for each item as it is processed. Well, sometimes it can be more efficient to make updates in bulk. Some databases support bulk or batch inserts that allow you to provide a whole set of updates and make a single call to the database to do them all at once. This can be much more efficient, especially if there is some network latency or other overhead involved in getting to the database.

Of course, you don't have to do that. Your writer can choose to just iterate over the `List` of objects and write them one a time to wherever you are writing things. Whatever is most convenient.

We've waved our hands a lot around the idea of a checkpoint. Next time we'll look at some different ways to decide when they should happen.

Ka-Chunk – What's that? – Ka-Chunk – I think I heard something..

Yup…it is time to talk about the `ChunkListener` (I know, I know, but it is hard to come up with interesting titles to these).

In earlier posts we've talked about the Job and Step listeners. Lately we've been talking about a chunk step and so it is time to talk about the Chunk Listener. The Step Listener still gets control around the whole step, but the Chunk Listener is a chance to get control around each checkpoint. We still need to talk about how checkpointing works. Maybe next time, if I can think of a clever title (don't hold your breath).

Like the Job and Step Listeners the Chunk Listener has methods that get control at the beginning and end of each chunk. These are called, not surprisingly, `beforeChunk` and `afterChunk`.

Understanding the timing of these is important. Both before and after methods are driven *inside* the chunk. That is, the transaction that wraps the processing for each chunk begins before the `beforeChunk` method is driven and the transaction is committed *after* the `afterChunk` method. That means anything you do inside these methods that is transactional in nature will get committed along with the chunk transaction (or rolled back if that's what happens).

There's also an `onError` method. If there is an exception in processing the chunk that isn't handled in some other way, the `onError` method will get control instead of the `afterChunk` method. The `onError` method gets passed the exception that is causing the chunk to roll back and `onError` gets control before the rollback occurs.

That means you can tell whether a given chunk will commit or rollback based on whether `afterChunk` or `onError` in the Chunk Listener gets control.

How is that useful? What might you want to do in a Chunk Listener? Honestly, I haven't really run across any common use cases, but I'm sure there are some. You could use the `afterChunk`/`onError` distinction to provide some sort of status update to someone monitoring the job. Maybe update something about how many chunks were committed or rolled back in this step. Although that information is also available through the `JobOperator` interface as part of the Step Execution object. Maybe you want to push the data somewhere instead of having to ask for it.

Other ideas? Let me know….

To Checkpoint Now, or Not To Checkpoint Now, and if Not Now, When?

Let's cover the easy stuff first. If you don't specify anything, the default is to checkpoint after every ten items are processed. That means the chunk reader and processor will be called to read and process ten items, the writer will be called with the results of processing, and the transaction wrapping all that will be committed.

Is ten the right value? Maybe, but probably not. Before we talk about how often you should checkpoint (here's a hint, "It depends.."), we should take a look at different ways you can control when a checkpoint is taken.

First of all, you can override the default value of ten items by specifying an item-count value in the chunk element of the JSL. You can also specify a time-limit (in seconds). That causes a check after each pass through the reader/processor to determine if the time limit for this chunk has expired and, if so, to commit. With a time limit you could easily process a different number of requests in each chunk, depending on how things are running.

I should mention a quirk of JSL here. You can specify both an item-count and a time-limit and they both apply. The checkpoint will happen after the count of items or the time limit has expired, whichever comes first. The default for time-limit is zero, which means no time limit applies. However, if you just specify a time-limit value, the default item-count value of ten still applies which might hit faster than your time-limit. If you just want a time-limit, you should probably configure a really large item-count value to keep it out of the way. The specification doesn't say what happens if you specify an item-count of zero, so probably safest not to count on that turning it off, even if some implementations behave that way.

Should you checkpoint often? Or not? In favor of frequent checkpoints is minimizing the amount of work that has to be re-done if a failure happens and work since the last checkpoint is lost. If locks are being held by chunk processing, smaller checkpoint intervals minimize the time those locks are held.

On the other hand, frequent checkpoints can add a lot to the elapsed time for the job. Suppose (and I'm just making this number up) doing a checkpoint takes half-a-second. For a job processing one million records doing checkpoints every ten items, the cost of checkpointing is (1,000,000/10)*0.5 = 50,000 seconds which is almost 14 hours. If we checkpoint every 1,000 records instead then it only adds 500 seconds. Again, I made up the half-second number, but you can see any cost multiplied out by a lot of checkpoints is going to add up.

Choose wisely…(or at least have a solid-sounding explanation for your choice :-)

Next time – writing your own checkpoint algorithm…

Go (or Checkpoint) Your Own Way

Apologies to Fleetwood Mac….

Last time we talked about configuring an item or a time based checkpoint interval. If neither of those work for you then perhaps writing your own checkpoint algorithm is the answer. Just specify a `checkpoint-algorithm` element that points to your Java class that implements the `CheckpointAlgorithm` interface and you're off.

The `CheckpointAlgorithm` allows you to get control at some interesting points in chunk processing and gives you a couple of options for controlling the checkpoint interval.

First of all, there are the `beginCheckpoint` and `endCheckpoint` methods which get control around the processing for an individual chunk. It gives you a chance to get control before and after the checkpoint. The `endCheckpoint` method gets control after the commit happens so you are outside of the transaction. The `beginCheckpoint` is, likewise, before the next transaction starts so you are outside of the chunk transaction here too.

To control deciding when to checkpoint, the interface includes a method called `isReadyToCheckpoint`. This gets control after each pass through read/process of an item and returns a `Boolean`. Basically, this gives you a chance, after each item, to decide if you want to checkpoint now or not. Since this gets control on every pass through the loop, don't do anything slow or expensive because the cost will add up. And remember you are inside the transaction so if you do touch a transactional resource in this method it is included in the transaction.

Finally, there is the `checkpointTimeout` method. This gets control before every chunk starts and allows you to set a time interval, in seconds, for this chunk. This is just like setting the time-limit in the JSL, except that you can choose a different value for each chunk.

So how would you use this? The most likely approach is to count the items (when you get control in `isReadyToCheckpoint`) and watch the time between passes through read/process and adjust the count or the time when you checkpoint based on how things are going to achieve whatever your goal is.

The tricky part isn't implementing an algorithm to achieve your ideal checkpoint interval. The tricky part is figuring what your ideal checkpoint interval should be….

Listen to the Rhythm of the Falling Rain (or the Chunk Processing…).

Apologies to The Cascades this time (had to look that one up..).

This time we're going to consider three Listeners you can implement around Chunk processing. We already talked about the Chunk Listener that gets control around the entire Chunk, but there are also listeners around the individual read, process, and write operations.

The Read Listener gets control before and after the `ItemReader` reads an item to process. The after method gets passed the object that was returned by the reader which gives you an opportunity to look it over and do any post-reader processing you might need to do.

The Processor Listener gets control before and after the `ItemProcessor` processes the item that was read. The before method gets the object returned by the reader, giving you a chance to do any pre-processing work. The after method gets both the object that was read and whatever object was returned by the processor. This gives you a chance to look at both the input and output from the processor.

Finally, the Writer Listener gets control before and after the `ItemWriter` writes the list of items returned by the processor in this chunk. The before and after methods both get the list of items to write. This gives you a chance to do something with the list both before it gets written and afterwards.

But that's not all! All three of these listeners also have a method that gets control if an exception is thrown from their respective artifact (so the `ItemReadListener` has an `onReadError` method that gets control for an exception from the `ItemReader`, etc.). All the methods get passed the exception thrown. This means you can create an exception object and use it to communicate between the reader/processor/writer and the listener. The processor and writer listeners also get the item (or items) being processed/written.

Be careful though…the `onError` methods are not catching the exception. They just get informed that it happened. Handling the exception is something we'll get into in future posts.

So, what good is all this? Probably you've tried to create somewhat generic readers, processors, and writers that do specific tasks (read from this data source, do this processing, etc). The listeners around them give you a chance to smooth out the edges between them or do some specific processing for this job that you don't want in the general use reader etc. As with most listeners, it is just a chance for you to get control around the mainline processing and do some extra stuff.

Five Little Monkeys, Jumping on the Bed…. Skipping in Java Batch

As a chunk step works through wherever the `ItemReader` is reading from, it might run into the occasional record that has something wrong with it. Maybe account numbers are supposed to be numbers and it finds one with a letter. Maybe the account holder's last name field is blank (and it isn't Madonna's or Sting's account). Or some other thing is wrong with the record. Should you fail the job right there? Probably not. One (or two, or three) bad records isn't enough to stop the whole job which is possibly processing millions of records that are just fine. You just want to skip this one and move along.

The first part of skipping records is coding the `ItemReader` to throw known exceptions when records are bad in a way that makes it one you would like to skip. Your application can define classes that extend the generic `Exception` class and call it whatever you like. Maybe `NonNumericAccountNumber` or `MissingFamilyName` or even just `BadRecord` with some specifics in the `Exception` class about what went wrong. You should include information in the exception about the problem record (which record it was or other identifying information). We'll see why in a moment.

Next you need to update the JSL for the step to define these exceptions as expected errors for which the record should be skipped. Any time the `itemReader` throws an exception, the batch container will compare it to known skippable exceptions defined by inclusion in the `skippable-exception-classes` element in the JSL.

When a skippable exception is thrown by the `ItemReader`, the batch container doesn't call the `ItemProcessor` (since there is no item to process) but instead just calls the `ItemReader` again to read the next record. The `ItemReader` needs to be smart enough (remembering which record it is on) that, having thrown an exception on the previous record that it knows is skippable, it will proceed to read the next record. We'll compare this to retryable exceptions later on.

But before the reader gets called again, any `SkipReadListener` defined for the step will get called. The `SkipReadListener` gets passed the exception that was thrown by the `ItemReader`. If you remembered to include information about the bad record in the exception, the skip listener can log information about the bad record somewhere to get the record corrected. Maybe it sends an email to the owner of the table data.

There are also `SkipProcessListener` and `SkipWriteListener` interfaces you can implement in case you throw an exception from the processor or writer that is defined as skippable in the JSL.

‘Round and ‘Round – Retry Processing

A search suggested “Try Again” by Aaliyah or maybe “Coming Around Again” by Carly Simon, but I decided to go with “Round and Round” by Ratt. Trying again is a common theme in music so I’m sure there are a lot of others.

Last time we talked about using Skip processing to skip over a problem without failing the whole job. This time we’re going to look at handling errors via retry processing. Basic retry processing is, as you would expect, just having another try at doing something that didn’t work the first time, but you think might work if you have another go.

The processing is pretty easy. Let’s consider a retryable error from an `ItemReader` (processing is similar for retryable errors from the processor or writer). Your reader tries to read a record to be processed and fails. It isn’t something wrong with the record and it isn’t something you consider fatal. Fatal might be something like the file you were reading from is suddenly gone. A retryable error might be something like a temporary connection failure to a remote database. The database might be down, but it also might just be a glitch in the network. We should try again at least a time or two.

In this case the application code in the reader throws a specific exception you’ve defined for this situation. Syntax in the JSL indicates this exception is a retryable error.

Side Note: There are both retry and retry-rollback exceptions and the syntax in JSL gets a bit wonky. We’ll look at that later. For now we are just looking at how retry processing works, not how you ask for it to happen.

What happens now? First of all, the `RetryReadListener` (if you have one) will get control in its `onRetryReadException` method and get passed the exception your reader threw. I’m not sure what you would do here. Maybe there is some check you to do to see if the remote resource is still there, or at least log a warning about the failure and that you are trying again.

And then the reader just gets called again to try again to read the next item. There is a maximum number of times you can retry reading records (and a maximum skip limit). Both are specified in the chunk element and default to unlimited.

What if you have both skippable exceptions and retryable exceptions thrown from the reader? When the reader gets control how does it know if it is supposed to retry reading the same record or just read the next one? You have to have some flags or counters or something local to the reader (although maybe accessible to the Listeners) to help the reader figure out what it is supposed to do. It threw the exception, so it knows what happened and needs to adjust so it does the right thing when it is called again.

Get Comfortable and Go Back In Your Mind with Retry-Rollback Processing

Last time we talked about basic retry processing. In that scenario we had a problem and simply went back and tried to process the same record again. This time we're going to go back farther. Relax and watch the spinning pendant….back you go…farther and farther…wait..not that far…just to the last checkpoint!

(ok, so…song title…. I'm going with "Back in Time" by Huey Lewis, but I'm sure there are others)

Right, so retry-rollback processing undoes whatever we've done since the last checkpoint and then tries again. We'll close the reader and writer and rollback the current chunk transaction. That will undo any transactional things we've done in this chunk. Of course, if you aren't making transactional updates (maybe just writing things at the end of a file) you'll need to manually go back and undo whatever you did – which might mean retry-rollback processing isn't for your scenario.

After the rollback, the open method for the reader and writer will be called again. Remember that these methods get provided the checkpoint data from the last checkpoint which should enable them to reposition themselves to take up where they left off.

Here comes the weird part. Chunk processing begins again with a call to the reader and the processor, but then we call the writer and immediately checkpoint. No matter how you have configured your job to checkpoint (item based, time based, or via an algorithm) on retry-rollback checkpointing will occur with an item-count of one until we get back to where we failed.

Suppose you had a job with item-count=10 and failed processing item number 13. There was a checkpoint at item 10 and the JSL says to retry-rollback this exception. The chunk will roll back to the checkpoint at item 10 and then read/process/write/commit for item 11, then item 12, then item 13. Once we get past the item that caused the retry-rollback before, then things return to normal and we'll checkpoint again 10 items later (item 23).

Be careful of this if you are expecting some specific number of checkpoints from the job. Suppose you have 100,000 items and checkpoint every 1000 items. You would normally expect 100 checkpoints. But if you fail 500 items into a chunk and have the job configured to retry-rollback that failure, you will have 500 checkpoints just getting back to spot of the failure.

The reason for this behavior is to allow you to 'sneak up' on the problem record and try to commit as many good records as possible before you hit the bad one again.

When I snap my fingers, you'll wake up and have a great day….1….2….3…..<snap>

JSL Syntax for Retry with and without Rollback

Wow..this sounds really really boring. I'm writing it and I don't really want to read it.

For a song title, I'm going with "I'm gonna sit right down and write myself a letter", recorded by a lot of people. You'll see why when we get to the end.

Well, the syntax is a bit goofy and certainly not what I would have thought was an obvious way to handle it. So, it seemed worth spending one post just to go through how this works. Likely all you will remember from reading this, if you do, is that the syntax is goofy and if you decide to try to do retry and retry-rollback processing, you should look it up. Good idea. If you remember that, you got the point. You can probably stop reading now. If you even got this far.

Ok, so to specify that you want to do retry processing with a rollback you have to specify what exception(s) you want to handle that way. Here's an example (stolen from the spec):

```
<retryable-exception-classes>
     <include class="java.io.IOException"/>
     <exclude class="java.io.FileNotFoundException"/>
</retryable-exception-classes>
```

That says that any `IOException` should trigger retry-rollback processing unless it is a `FileNotFoundException` (or any subclass of `FileNotFound`).

Ok, well enough. We've been talking about retry-rollback so it seems a bit odd that the JSL clause just says retryable and nothing about rollback, but ok, fine. But then…how do you specify an exception that you want to just do retry processing without the rollback? That's where it gets weird.

The `retryable-exception-classes` element specifies exceptions that will retry. Maybe with rollback and maybe without rollback. What determines which one is whether or not the exception is ALSO listed in the `no-rollback-exception-classes` element. Suppose we wanted to do rollback processing for `IOExceptions` (except `FileNotFound`), but not do rollback processing for an end-of-file exception? Then we need to ALSO specify this in our JSL:

```
<no-rollback-exception-classes>
     <include class="java.io.EOFException"/>
</no-rollback-exception-classes>
```

Because the EOF exception is a subclass of `IOException` it will be included in the exceptions for which retry processing will occur. Because it is listed in the `no-rollback-exception-classes` it won't do rollback, but all the other `IOException` subclasses will.

As you can see, this is a bit tricky. If you aren't doing something with subclass exceptions, you can end up listing the exact same exceptions in both clauses. Suppose we just wanted to do retry processing (with no rollback) for any `IOException`. Then you would need this:

```
<retryable-exception-classes>
     <include class="java.io.IOException"/>
</retryable-exception-classes>
<no-rollback-exception-classes>
     <include class="java.io.IOException"/>
</no-rollback-exception-classes>
```

Just listing an exception in the `no-rollback-exception-classes` element does nothing at all. That element only applies to exceptions (or parent class exceptions) that are specified in the `retryable-exception-classes` element.

Bottom line: If you decide to use retry processing, with or without rollback, be sure to put some nice comments in your JSL to explain what you intended. The year-from-now-you will appreciate it when you're puzzling over it.

A review of Chunk Step Listeners

There are a lot of songs about listening. Chunk processing has a rhythm to it, so I'm going with "Listen to the Music" by the Doobie Brothers.

A batchlet step just has the basic Step Listener, but a chunk step has a bunch of different listeners you could use. I thought we might wrap up our coverage of chunk steps by taking a quick look through them all.

First there is the chunk listener itself which gets control before and after each chunk, plus when any unhandled exceptions are thrown. This is your opportunity to get control around the transaction that wraps each chunk of processing.

As we dive inward into the chunk loop we also have listeners around each piece of the processing in the chunk. The `ItemReadListener` gets control around the `ItemReader`. The `ItemProcessListener` gets control around the `ItemProcessor`. The `ItemWriteListener` gets control around the `ItemWriter` processing. These listeners give you the opportunity to get involved before, after, and in between the parts of chunk processing. They also get control if unhandled exceptions are thrown within the reader, processor, and writer.

Then come the skip listeners. There are three separate skip listeners, one each for the reader, processor, and writer. They get control if a skippable exception is thrown. Which one gets control depends on which artifact threw the exception. All of these listeners get passed the exception that was thrown.

This might be a good place to remind you that a single Java class can implement more than one listener because the method names are different. For instance, a single class could implement the `onSkipReadItem` method, the `onSkipProcessItem` method, and the `onSkipWriteIem` method. You would specify this Java class in the JSL as a listener and the batch runtime would realize it implements all three listeners and call the right method whenever appropriate.

Also recall that more than one class can implement the same listener interface. There is no rule about what order they get called in, but they will all get called when appropriate for the interface they implement.

We'll wrap this up with the retry listeners. There are three of these also. One listener is defined for reader, processor, and writer. The right listener will be called based on which batch artifact threw a retryable exception (whether it was a retry with or without rollback, there's just the one listener).

That's it for our discussion of chunk processing. What's next I wonder….

Introduction to Partitions

We've already talked about using Splits and Flows to run different steps concurrently within the same job. Now we're ready to move on to partitions. With a partition we're running multiple copies of the SAME step concurrently.

They run in parallel, so we'll go with "Parallels" by Yes.

The traditional example of this is a step that needs to do the same processing with a lot of records. Suppose you have a table with one record for each account. You need to look at every account and do some sort of processing and then put a result somewhere. Maybe you need to insert a new row into a different table for any account you find that matches some criteria.

A simple chunk step will iterate across every row, do the processing, and, at check-points, write the rows that met the criteria. If you have a lot of rows, this could take a long time. With a partitioned step you can break that up into several threads, running concurrently, each processing a different range of the data.

In a later post we'll look at how to decide how to break up the work and how to communicate with each thread what it is supposed to do. But for now, just assume there is a way and we can take our table of one million rows and process it with ten threads each concurrently processing one hundred thousand rows.

Of course, just because you want ten threads running concurrently doesn't mean they really will. Depending on the platform and available processors and a host of other factors, they might not run as concurrently as you'd hope. And we've also made some assumptions that these threads won't get in each other's way. We will have ten threads inserting rows into the same table at the same time. Is that too many? Will the contention doing the inserts outweigh the benefits of concurrent processing? Maybe.

Another thing to remember about partitions is that they can be batchlets instead of chunk steps. You could have a simple batchlet that copies a file and use a partitioned batchlet step to copy multiple files at the same time, each partition copying a different file.

Next time we'll have a look at the JSL syntax to define a partition and the simplest way to tell each partition what to do.

Static Partition Control

A song about control? What else, “Control” by Janet Jackson…

I’ve tried to avoid spending posts going through syntax details. You can get that from the specification. But the best way to explain static partition control is by looking at the syntax of specifying a partition in JSL. Try to stay awake. Next week we’ll look at writing code to dynamically define the partitioning. Doesn’t get more exciting than that…

A step becomes a partitioned step by including the `<partition>` element inside the `<step>`. It looks, roughly, like this:

```
<step id=”step1”>
     <chunk >
          ..chunk stuff goes here
     </chunk>
     <partition>
          ..partition stuff goes here
     </partition>
</step>
```

There’s a lot of different ‘partition stuff’. This time we’ll just look at the partition plan. The plan is where we tell the batch runtime how many copies of the step (how many partitions) we want and what properties to pass to each copy. Here’s an example:

```
<partition>
     <plan partitions=”2” threads=”2” >
          <properties partition=”0”>
               <property name=”startRec” value=”0” />
          </properties>
          <properties partition=”1”>
               <property name=”startRec” value=”1000” />
          </properties
     </plan>
</partition>
```

There’s a lot in there. First of all, we’ve decided we’re having two partitions. We also tell the runtime that both of them can run at the same time on different threads. If we had 10 partitions, we might tell the runtime to only allow up to five of them to run concurrently (for some reason). The number of threads doesn’t guarantee you will get that many, just that you won’t get more partitions running concurrently than that. You might want to break the data up into 10 sets but know that contention causes issues with more than five running concurrently.

Inside the plan we specify the properties to inject into each copy of the step. In our case the `<reader>` element in the chunk probably has a property whose value is set to something like "`#{partitionPlan['startRec']}`". That says that on each partition thread the copy of the reader running on that thread will have a value injected for startRec that comes from the partition plan for this partition.

Note that the partition numbering starts with zero. Be sure to provide property values for each partition. You also need to be sure each partition gets all the information it needs. In our example, how does partition zero know when to stop? It starts with record zero, but it doesn't see the start value of '1000' that the other partition gets. Is it hard-coded to only process 1000 records? Or do we need to provide a stopRec parameter to each partition also?

Defining your partition this way is fine if you know ahead of time what the right partitioning is and what properties to provide to each partition. A step that copies multiple files might be a good example because you probably know ahead of time which files get copied at this point in the job. Scenarios with record numbers tend to be less static (what table will have the same number of rows – forever?). For those cases you should use dynamic partition definition which we'll look at next time.

Creating a Partition Plan Dynamically with a PartitionMapper

A song about dynamic partition planning. That's a tough one. Maybe a song about breaking up? Perhaps "50 Ways to Leave Your Lover" by Paul Simon?

Last time we saw how to create a partitioned step just using syntax in the JSL. It is pretty easy to do and makes it clear how many partitions you want and what properties get passed to each copy of the step. Meanwhile, out here in the real world, you often don't know ahead of time exactly what you're going to want to do.

Fortunately, instead of specifying everything explicitly in the JSL, you can just point to an implementation of the `PartitionMapper` interface. The mapper has one method, `mapPartitions`, that returns a `PartitionPlan` object. That means you need to create your own class that implements the `PartitionPlan` interface.

The interface defines methods to return all the information you specified in the JSL: the number of partitions, the number of threads, and the properties to go to each partition. A `PartitionMapper` looks at the data it is going to process and makes some sort of decision about how to split things up. You might know ahead of time that 10 partitions are how many you want and determine the data ranges for each partition to process dynamically. Or you might know that you want each partition to process 100,000 records and decide how many partitions you need based on the amount of data you have, possibly restricting it to no more than 10 threads running your partitions at a time. These numbers are just examples. You need to determine what works best for you based on your application and your data.

How do you specify those properties for each partition? By creating an array of Java `Properties` objects, one object in the array for each partition. Then you can set whatever properties names and values you like into each `Properties` object and be assured that the value of property called `starting_record` in the zero-slot of the array will be given to the first (remember we start counting at zero) partition.

One gotcha…. Remember that names and values in `Properties` objects are `Strings`. Use the `setProperty` method to put values into the object. But because `Properties` inherits from `HashMap`, you may be tempted to use the `put` method to do something like this:

```
myProperties.put("start_record",1000);
```

You think that sets the `start_record` value to 1000, but when the property value is used, that isn't what you'll get because it has to be a string. Do it like this:

```
myProperties.setProperty("start_record","1000");
```

It is a bit awkward because you'll have it as a number in the `PartitionMapper` and want it as a number in the `ItemReader`, but you need to make it into a string to get it there.

How do I know this is a tempting thing to do? Well…..

Coding a Skippable, Retryable, Restartable, Partitionable Reader

How about "Listen, Learn, Read On" by Deep Purple?

An `ItemReader` only has four methods: `open`, `readItem`, `checkpointInfo`, and `close`. The `open` method opens a connection to a datasource or a file. The `readItem` method reads one record and puts relevant information into an object to be handled by the processor. The `checkpointInfo` method returns whatever serializable object contains your checkpoint information. And the `close` method closes the connection or the file. Seems pretty simple really.

And it can be, unless you want to be able to do skip or retry processing, or you want to restart a failed job, or you want to use partitions. Then you need to be a bit more careful.

Since we're on a stretch of posts about partitions, we'll consider this aspect first. A basic reader is just going to create a connection or open a file and just read until it runs out of stuff to read and then stop. A partitionable reader needs to know where to start and stop. For a file that means knowing which record in the file to start on and which one to stop on. For a database table, it assumes the records are sorted in some order and, again, it knows where to start and stop. All this information can be put in `PartitionPlan` properties that are different for each partition. When your reader is used in a simple chunk and not a partitioned one, choose defaults for the properties that make it clear you want to start at the beginning and end at the end.

To handle retry with rollback or a job restart (we haven't talked about restart processing yet), it all comes down to the checkpoint data. When the step starts for the first time, there won't be any checkpoint information, but on a restart or retry it will contain whatever you put in the checkpoint object at the last checkpoint commit. To pick up where you left off, you need to be sure to put whatever information you will need in that object. This is probably just the record number or whatever you are using to track where you are between the start and end of the range you are processing.

Handling skippable and retry without rollback scenarios is pretty easy if you have all that working. Before your reader throws an exception that it knows will be listed as skippable or retry-no-rollback, just be sure to adjust how you are tracking your current location so you do the right thing next time around (increment forward or backwards as necessary).

If you are trying to write a 'generic' reader from some datasource, give some thought to whether you want to support it being used in a partitioned step and plan ahead. Spend some time considering how it will handle skips, retries, rollbacks, and restarts. Somebody else might take your reader and use it in JSL they are writing. Make clear to them how it can be used and what it expects in terms of parameters and how it behaves on errors. Or just put your phone number at the top and expect to get a call at 3:00 in the morning.

Analyze This! The Partition Step Collector and Analyzer

Went with a movie title this week…. There's probably a song in here somewhere too.

Up to now we've talked about partitioned steps that were acting independently. Maybe a bunch of batchlets running concurrently to copy a list of files. Or more commonly a chunk step with each partition reading and processing a different range of a large data set. But what if the point of each partition is to build towards a single result for the whole data range.

Suppose we want a step that analyzes a large set of data looking for records that match a certain pattern (and assume this isn't something we can do with some clever SQL – because it makes the example work). Running without a partition, the chunk step will read every record, process it to determine if the record meets the criteria and increment a counter, and that's it. The writer doesn't do anything. When we run out of data, the result of the step is the final value in the counter. That's all we want.

To do this faster, we'd like to run this step using partitions. We set up the `Partition-Plan` so each partition knows what range of data to read and process. But as each partition ends, they each have their own result. How do we bring those together and add them up to the final answer we want? The Collector/Analyzer is the answer.

The `PartitionCollector` gets control at the end of each chunk. It doesn't get anything passed to it, so any result it needs to work with has to be hung on the thread or attached to the `StepContext`. The collector returns a serializable object, which can be anything you like. In our example it is probably an object that contains the count of matching records from this chunk (remember to reset the count after each chunk happens).

As chunks complete on the running partitions and the Collectors provide data, the `PartitionAnalyzer's` `analyzeCollectorData` method gets control back on the main thread for the step and is passed the serialized data provided by a Collector. This is where you would add the result into the running total for the whole step. A transaction is wrapped around all of the analyzer processing, so whatever it does will commit (or rollback) when all the partitions end.

The Analyzer also has a method called `analyzeStatus` that gets the final batch status and exit status values for each partition. This allows the analyzer to know that a partition has finished and how it turned out. If something went wrong with one partition, this is how the analyzer finds out. The batch status might be just fine, but something subtle could have happened that the partition can communicate to the analyzer via the exit status string.

Be careful with collectors and analyzers. If a chunk completes and commits, there is no guarantee the collector data from that chunk will make it to the analyzer if something bad happens. If you are trying to make a job using a collector/analyzer restartable, you should bear this in mind.

Partitions Too Fat? Get a Partition Reducer! (No, not really)

This is kind of a weird one. The `PartitionReducer` gives your batch application a chance to get control at some interesting points during the processing of a partitioned step. The intent was to give you a chance to do transaction-like things if your application was interacting with non-transactional resources (like a file). Let's have a look:

The first method in the reducer is `beginPartitionedStep`. This method gets control on the main thread for the step, before the `PartitionMapper` is called and, thus, before any partitions even start. This is almost a listener-like point that gets control before any partition stuff has happened at all, but you know it is going to.

The matching method that receives control when all the partitions are finished is called `beforePartitionedStepCompletion`. Why isn't it called `afterPartitionedStep`? Because it doesn't get control if any of the partitions fail. There are actually a few different ways this can happen, but basically if an unhandled exception gets thrown out of the partition processing (including from the `PartitionMapper`!) then this method won't get control.

After `beforePartitionedStepCompletion` gets control, the transaction that wraps `PartitionAnalyzer` processing on the step thread is committed. After the commit, another method called `afterPartitionedStepCompletion` is called. These two methods (before and after step completion) wrap the commit to let you do commit-like processing for any resources touched by the analyzer that enrolled in the transaction.

On the other hand, if one of the partitions fails, then once the last call to the analyzer is done, we're going to rollback the transaction around analyzer processing. But before that happens, the Reducer's `rollbackPartitionedStep` method gets control. So, either the before/after completion methods get control or the rollback method does, depending on how things went with the partitions.

This is all rather complicated, and you'll have to think through what your partition analyzer is doing to determine if you really even care how things turned out.

Restarting a Partitioned Step

We haven't gotten to how you restart a failed job and how all that works (soon), but while we're talking about partitions I thought I'd have a quick look at when job restart happens to a partitioned step.

Oh right, we were doing song titles to go with these. Let's see…a song about restarting after a failed partition (or a break up)? Must be thousands of those…take your pick.

The key thing is the partition properties set up by the `PartitionMapper`. Do you use the same properties at on the original run of the job or do you create new ones? If you are dynamically figuring out how to partition things and the underlying data might have changed since the original job ran, do you want to re-partition or stick with the original decision? Could you miss records using the original scheme? Do you care?

The `PartitionPlan` returned by the mapper tells the batch container what you want to do. The value returned from the `getPartitionsOverride` method on the `PartitionPlan` determines the behavior. If your plan returns 'true' then you are going to provide a whole new plan. You can have more or less partitions than you originally had. You can supply new properties. Basically, the step starts over. This is easy, but if your first attempt at this step processed (and committed) some changes you might be processing those records again. Maybe that's ok and maybe it isn't. Or maybe you can tell and set up the new partition properties to do the right thing.

If the partition override value is false, then you want to pick up where you left off in the last try. Each partition, if it didn't complete, will start using the checkpoint data from the previous run. However (and this is a big deal), your plan MUST be set up to return the same number of partitions as it did the previous time and the same properties assigned to the same partitions. That means if the prior run gave partition 0 a property of ABC="1" and partition 1 a property of ABC="2" then those same partitions have to get those same properties.

The property values from the previous run aren't automatically provided to the partitions. You have to remember what you did and do the same thing again. If you are dynamically determining how to partition things, and the environment might have changed, this could be challenging to get right.

Play it again, Sam! Restarting a failed job

First of all, yes, I know that's not really a quote from Casablanca, but the actual quote ("Play it once, Sam") doesn't work as a title for a post about restarting jobs.

Before we get into the details of how job restart processing works, we need to get our heads around a few basic concepts. Then we can look at what makes a job restartable, and how you restart it.

To begin with, when you start a job it creates something called a Job Instance. This represents the submission of this job at this time. If you submit a job every Tuesday at noon, then every Tuesday at noon you get a new Job Instance. As the job starts to run, it creates a Job Execution. A Job Execution represents an attempt to, ahem, execute a particular Job Instance.

Which means, if a job fails and you restart it, you get a new Job Execution for this Job Instance. If you give up and just submit it again, then you get a new Job Instance with its own Job Execution. Multiple executions for a single instance mean you have re-started that job.

To restart a job, it has to be restartable. I know….seems obvious, but what does that mean exactly? Well, first of all it has to be in a state that makes it restartable. That means the job was started, created an Instance and an Execution, and the Execution's batch status is either STOPPED or FAILED. A job can only get into those states if it was stopped using the Job Operator Stop operation or if it threw an unhandled exception.

There's an attribute of the JSL that matters too. In the `<job>` element in the JSL there is a *restartable* attribute that defaults to true. If set to false, then the job cannot be re-started regardless of its state.

The actual act of restarting a job requires calling the `restart` method (instead of the `start` method) on the Job Operator API. Restart requires you to specify the identifier of the execution you wish to restart. This has to be most recent execution id for a job instance. Which means, if a job fails and is restarted multiple times, each restart has to specify the execution id of the previous failure. You can't just restart a job instance, you have to restart an execution after it has failed. Which means you can only restart a particular execution once. After that you get a new execution and, if it fails, you have to re-start that one.

That's enough to get started on restart..

Job Restart Processing – How it works..

In this post we'll take a look at the mechanics of restart processing. When we wave our hands around and talk about restarting a job, we generally sort of suggest that it just "picks up where it left off" which is, of course, wrong.

For a song I'm going to go with "Get Right Back" by Maxine Nightingale, a suggestion for an earlier post from a loyal reader.

In general, a restarted job starts over again with the first step of the JSL (unless the failing job execution sets a restart step when it ended with a <stop> element). But that doesn't mean it actually runs that step (or any other step). The key is an attribute of the step element called `allow-restart-if-complete`. If set to true, then the step is re-run, even though it completed successfully on the previous run. This sets a new value for the exit status of the step and flow proceeds based on what is in the JSL conditional flow for the step and the new exit status.

On the other hand, if `allow-restart-if-complete` is false and the step completed on the previous execution then whatever exit status resulted from that run is used. Put more simply, we skip this step and proceed to whatever step we flowed to last time.

That all means that if you always have `allow-restart-if-complete` set to false or it is true for some steps, but they produce the same exit status on a new run, that flow through the job will proceed exactly as it did the previous time. But if a step is re-run and produces a different exit status on this run it is possible the flow could proceed to a different step. That step might not have run at all in the previous execution. In which case it has no saved exit status and must be run for this execution.

That's all great for steps that completed in the previous execution. But eventually you might reach a step that was executing when the job stopped or failed. That step is always run as part of the restart of the job (assuming flow of control reaches that step).

If the step is a partitioned step, then the individual partitions are restarted as we discussed a few weeks ago.

We'll talk about how a chunk step gets restarted in a separate post.

Restarting a Chunk Step

We've touched on this in earlier posts, but since we're covering all the restart processing lately, I thought it was worth going through again. The idea is that you have a batch job that was running a chunk step when the job was either stopped or it failed (threw an unhandled exception). When you restart the job, processing goes through earlier steps in the job as we discussed last time and ends up back in the chunk step. What happens now?

The checkpoint data is retrieved from the Job Repository for the step from the last execution and the `open` method for the `ItemReader` and `ItemWriter` are called, providing the serialized checkpoint data. This is exactly like the processing that `open` has to do in handling a retry-with-rollback scenario. The reader and writer both have to orient themselves, so they are ready to pick up where they were at the last checkpoint. This might mean getting a database cursor positioned properly, or maybe getting positioned in a file at the appropriate record. It depends on the data source being used.

Once that is done, processing proceeds as normal for a chunk step. Unlike the retry-with-rollback scenario, we do not "creep up" on the failing record one item at a time. A restarted chunk step just uses whatever checkpoint interval or algorithm is normal for the step.

While a step can have a limit on the number of retries (or skipped records) it will allow before failing the job, a failed job can be restarted any number of times.

Remember that for a partitioned step containing chunk processing, each partition will normally go through this exact processing. Although you can change that by choosing to redefine how the partitioning is done.

If the job didn't fail doing write processing and your reader handles receiving checkpoint data appropriately, restart processing is almost magical. But if the failure occurred during write processing and the writes aren't to a transactional resource, then it can get complicated. In this case the `open` method for the writer is going to have to undo updates that were made after the last checkpoint because they won't have rolled back automatically.

Being able to restart a job, and especially a job with chunk steps, can be critically important. Carefully consider what your application is doing, what resources it is accessing, and how your code will react when called in a restart (with checkpoint data) to be sure it does the right thing. Operations will tend to assume a failed job can be restarted. Don't surprise them….

Naturally our song this week is "Surprise, Surprise" by the Rolling Stones.

Job Parameters when Restarting a Job

The `restart` method on the `JobOperator` interface allows you to provide a `Properties` object containing restart parameters. This is equivalent to the `Properties` object of job parameters that can be supplied when a job is started for the first time.

The batch specification says that job parameters are NOT remembered from one execution to the next. That means that if you specified parameters when you started the job the first time, you need to decide if you want to specify those same (or different) parameters when you restart the job when it fails (or, to be optimistic, **if** it fails).

Whether you specify different values for the parameters on a restart than you did when you initially started the job depends, of course, on what those parameters do in the JSL. The parameters might point to files that need to be processed and you still need to process those same files.

On the other hand, it could be that the job failed because of a problem with some of those files and you have new, corrected, files somewhere else and the job restart needs to point there instead.

There is one thing you cannot change across a restart. If you have a partitioned step and coded the partition plan right in the JSL (meaning no `PartitionMapper` is involved), then the number of partitions must be the same. If the value used for the number of partitions comes from a job parameter, then you have to specify the same value on the restart as you did on the initial start of the job.

This post is shorter than most of them because this topic is really pretty simple and straight forward. I feel a bit guilty. Maybe I will just make it look longer and see if anybody notices. Obviously our song is “Killing Time” by Triumph.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non provident, sunt in culpa qui official deserunt mollit anim id est laborum.

And that about wraps it up for this topic.

The Job Operator Interface

We've mentioned the Job Operator interface before. This is the API that allows an application to start, restart, and stop jobs. Coming from a mainframe background, it always seemed odd to me that you would want to have a running application program call an API to start a batch job. Batch jobs are something run on a schedule with some sort of enterprise scheduler calling the shots. And some of the implementations of JSR-352 have provided extensions to the specification that allow you to do just that.

But your enterprise scheduler could easily start a JVM to run an application that, in turn, runs a batch job inside that JVM. It looks a little different to operations than their "normal" batch job, but it is a batch job all the same.

The best use of the Job Operator API, I think, is to spin off batch processing from OLTP applications. Suppose your banking application is processing an ATM cash withdraw and it needs to launch some fraud detection processing. That processing needs to happen asynchronously to the transaction and is, perhaps, complicated. The transaction could use the Job Operator API to launch a batch job within the same server that is processing the ATM transaction to run through the fraud detection process. Or something like that.

Job Operator also provides a whole set of APIs to allow you to poke around in the current batch environment. You can get lists of running jobs or lists of jobs that have already run with a particular job name. You can find all the executions of a particular job instance (attempts to restart a particular job). You can even access to the StepExecution objects for the steps within a job.

Another method allows you to access the job parameters used to submit a job execution. That's actually pretty important. If you are writing code to restart a failed job, remember that the spec requires you to provide job parameters for the restart. You can use the getParameters method to fetch the parameters used for a prior execution of a job you are about to restart, giving you a chance to adjust them, if necessary.

Remember also that having access to the Step Execution object allows you access to the Persistent User Data for the step. For a running job, this is the last persisted user data (at the last checkpoint if it is a chunk step). If your application is leaving "notes" about its progress in the user data, this is a way to get a peek at it from outside the job.

There are probably other choices for a song, but I'll go with "Smooth Operator" by Sade.

Try To Remember – The Job Repository

The Job Repository is an interesting aspect of the JSR-352 specification in that it is a critical piece of the ability to process jobs, but it is barely mentioned by the spec. The specification basically declares that you have to have one, but the details are outside the scope of the specification itself.

The purpose of the Job Repository is to remember things about the jobs you have run in the past and the current state of jobs running right now. The batch and exit status of every step of every job you have ever run is kept in the repository so you can go back and see what happened. The fact that some job execution failed has to be remembered so that you can restart it. The results of each step executed as part of that failed job has to be remembered so that restart processing can do its thing.

Jobs that are running right now keep their status up to date in the repository in case of a failure. This is especially important for checkpoint data that might be needed in the event of a failure and restart or even a retry-rollback scenario within execution of the step. For partitioned steps, the state of each partition (and information about how many partitions there are) have to be remembered.

But the specification leaves it up to the individual implementations to decide how to keep track of all this, in a transactional way. This likely means some sort of traditional database will be underneath. But the spec nicely leaves the way open for future developments of technology that meet the requirements in some new way.

Unfortunately, as a side-effect of not specifying how the Job Repository is implemented, the spec is also silent on how you get rid of things in it. It is just great that the repository remembers all the details about that job you ran two weeks ago. But eventually, unless some government regulation requires you to keep this sort of thing forever, you will likely cease to care about details of job executions from years and years ago.

Eventually the space required to store all of this information might become a problem. And you will want to remove some things from the repository (if just to make the results returned by APIs looking for information about a particular job you run every day a bit smaller). And the specification doesn't say. So…as part of evaluating how a particular implementation of JSR-352 might work for you, consider how they manage the repository and how you get things removed from it.

Our song is obviously "Try to Remember" which was originally sung by Jerry Orbach (yeah, the guy from Law and Order). You can find it online..have a listen.

Moving on….to WebSphere Liberty

Well, at least for the moment, I've run out of things to write about the JSR-352 specification. I'm sure there are some other things I've overlooked or oversimplified that might be worthy of another post. Feel free to suggest some things in the comments. But for now, I'm drawing a blank here.

So, I thought I'd move on to talking about the implementation of JSR-352 that I'm most familiar with. That would be the implementation in IBM's WebSphere Liberty. We added support for JSR-352 as part of the Java EE 7 support and it is fully spec compliant.

But we also added a lot of extra operational enhancements to help you write, run, and manage batch applications running inside Liberty. Some of those are pretty interesting (and, I think, useful). I thought I would spend some time looking at some of the enhancements and why I think they are cool.

I've tried to be pretty implementation neutral when talking about JSR-352, but, at least for the next series of posts, I'm going to show some pretty significant bias towards the Liberty implementation.

I'm planning to talk about some of the extensions and enhancements in Liberty that allow you to better integrate JSR-352 batch applications with an existing enterprise batch environment.

Since a lot of existing batch applications are running on z/OS, there are some Liberty Batch extensions that are specific to z/OS and help it better integrate into a z/OS environment (think SMF records). We'll have a look at those and perhaps also some of the challenges of mixing traditional (COBOL) batch applications with "modernized" Java Batch.

And after that…. Who knows?

Job Repository Implementations – Part 1

When we talked about the Java Batch specification, one thing we discussed was that the spec doesn't say exactly how an implementation keeps track of information about jobs it is running (or jobs that have run in the past). It isn't just information about whether the job was successful or not. An implementation has to remember checkpoint information for any chunk steps as well as the completion status of any steps that were run so it can do proper restart process if the job fails.

Since this is the first post in a series discussing the implementation of Java Batch in WebSphere Liberty, I thought I'd start off by talking about how the Job Repository is implemented by Liberty.

Actually, we offer some options about implementation. If you do nothing at all in the server configuration (except configure the batch feature, of course) then, by default, the Job Repository will be implemented using in-memory structures.

On one hand, this is really cool. You don't have to set up a database. You don't have to do any configuration. It just works. And if you make a mistake or do some things you'd rather just forget happened (and we've all done that…) you can just restart the server and the whole repository contents will be thrown away. This is just awesome if you are doing development or just playing around with batch.

Do remember that all the information, including serialized checkpoint objects, is being kept in memory. That takes up room in the heap. If you leave the server up for a long time and run a lot of jobs, information about old jobs could be essentially a memory leak. Purge some jobs or restart the server to tidy up.

Of course, if you actually care about the jobs you are running and want to be able to reliably restart jobs when bad things happen, you want a more persistent repository. Fortunately, Liberty provides a real persistent option that we'll talk about next time.

Since this week was mostly about the in-memory Job Repository, our song is going to be "The Way We Were" (Barbra Streisand). Jobs that are too painful to remember, you simply choose to forget – by restarting the server.

Job Repository Implementations – Part 2

Last time we talked about the in-memory Job Repository implementation offered by the WebSphere Liberty implementation of Java Batch. In memory is great and easy to set up (nothing to do at all). But for real jobs that you care about, you want a persistent store to keep that stuff.

The Liberty Batch implementation uses Java Persistence API (JPA) to access the database of your choice. JPA is pretty cool because it means that the batch support in Liberty doesn't have to have code that addresses all the different quirky differences between databases. That's all handled in the JPA layer.

It also means that WebSphere doesn't have to supply DDL for you to use to set up the tables that will be used as the repository. Supplying different DDL for different database implementations is an endless source of problems. With JPA you can actually let the runtime create the tables itself on the fly. The first time the batch code tries to access a repository table, the JPA code will realize the tables aren't there and go create them for you.

If having tables automatically created makes your skin crawl (or gives your DBA the willies) then you can also use a utility to just generate the appropriate DDL for your database. Then your DBA can give it a long gander before running it during an appropriate change window.

Recall that when you configure the server and want the in-memory repository you don't have to do anything. If you want a persistent database used as the repository you have to tell the server where it is. That means setting up the batch persistence configuration element in server.xml to point to a datastore configuration.

Or you can just set up the default datastore configuration and the batch code will find it automatically and use it.

Using the in-memory repository will ultimately result in job information piling up in the heap. With a persistent repository job information piles up on disk somewhere. Eventually you will want to purge information about some of those jobs too. We'll get to how purge operations work when we talk about the REST interface. That might be next…

For our song this week I think it should be "Memory" from CATS. When your job fails, you can go to the persistent repository and look at last week's job that worked.. "I remember the time I knew what happiness was, let the memory live again."

The REST Interface

Back when we were just discussing the JSR-352 specification we went over the `JobOperator` interface. That's the API the specification provides to allow you to start a job, stop a job, restart a stopped/failed job, and get information about running jobs and jobs that aren't running.

The `JobOperator` interface is great if you have running application code that needs to spin off some batch processing as part of whatever the application is already doing. But in an Enterprise environment you are likely to have some sort of batch scheduler that submits production jobs automatically. How would something outside of a server providing Java Batch capabilities get jobs started?

Well you can certainly invent your own way to drive work into the server and call `JobOperator`, but we thought we would help you out with that by providing a REST interface for batch. The REST interface essentially (although not actually) wraps the `JobOperator` interface and allows you to do all the same things, plus some extras.

The exact operation you want to perform is controlled by the REST verb (GET, POST, PUT, DELETE) and the URI used to access it. You can also provide parameters to the specific service as a JSON string parameter on the REST request.

We haven't talked about security yet, so for now we'll just say that the REST interface is protected so not just anybody can submit or manage batch jobs.

In subsequent posts we'll take a look at some of the specific capabilities of the REST interface, although we won't be going into intense detail. The Knowledge Center provides complete documentation about the API (in all its versions).

A great way to explore and learn about the batch REST interface (and any other enabled REST interfaces in Liberty) is to use the apiDiscovery feature. This makes any REST interface defined with appropriate Swagger documentation accessible via a browser. Just direct your browser at https://host:https_port/ibm/api/explorer for the appropriate host/port of your server and explore the available REST APIs. You can use it to drive the interface and get the exact required syntax (appropriate for use via cURL, if you like).

JobOperator Functions via the REST interface

In talking about the batch REST API we often wave our hands around and make it sound like it is just a wrapper around the `JobOperator` API. But it isn't. In fact, it is an entirely separate interface that happens to include some of the same capabilities as `JobOperator`.

`JobOperator` has a whole set of methods that allow you to get back information about jobs known to the Job Repository. You ask for it in various ways: by jobname, by instance id, and by execution id. In general, these methods return objects. If you ask for the Job Instance for a specific job execution id you are given the `JobInstance` object for that job. The object implements the `JobInstance` interface with a set of available methods.

The REST API has similar GET operations that let you access instances, executions, and step executions. Instead of returning objects, the REST interface naturally returns JSON strings that include the information about the returned thing that you would have gotten via the object returned by the `JobOperator` API. Thus, instead of a `JobInstance` object you would get a JSON string with all the information we have about the Job Instance.

The REST interface has a lot of filters that aren't available through the `JobOperator` API. You can search for jobs by id, of course. But you can also search by created time, last updated time, batch status, exit status, application name, submitter id, and even for job parameters with a particular value. That last one allows you to find all the jobs which were submitted with a particular name/value parameter pair such as JOB-CLASS=A.

The REST API also provides equivalent functions for the more operational aspects of `JobOperator`. These include the start, stop, and restart functions. The REST API does not provide an equivalent to the `JobOperator` abandon function.

Finally, there's a really important feature of the batch REST API that isn't available to `JobOperator`. We'll talk about that one next time.

Purging Old Jobs

In an earlier post we talked about the Job Repository that is used to keep track of the state of jobs currently being run as well as jobs that have finished. The specification doesn't get into any details about how the repository should be implemented. It just says you have to have something that performs these functions. We've also talked about how the Job Repository is implemented in WebSphere Liberty.

One thing the specification doesn't discuss is how you get rid of things in the Job Repository. Because it doesn't describe how the repository works, it can't tell you how to get rid of things. An implementation could choose some automatic archiving system. Or maybe it is just a hole in the implementation that there is no way to get rid of things. This is another one of those questions to ask when looking at an implementation of JSR-352.

The ability to purge jobs is one of the capabilities of the batch REST interface. It has no equivalent in the `JobOperator` API because the specification doesn't discuss job purging.

The DELETE verb is used to purge jobs. The syntax for purge requests is exactly the same as GET requests we talked about last time that let you filter jobs by all sorts of criteria. You can delete all the jobs with a certain application name, with a certain batch status, that were created before a particular time, etc. You can even delete jobs with specific job parameter values.

I'd like to emphasize that the filters you can use to list jobs is the same as the filters you can use to purge jobs. That means that you can use a search to get the list of jobs that match a particular set of filters BEFORE you use that same set of parameters to purge those jobs. It gives you a chance to be sure you got the syntax just right before you accidentally purge all the jobs SINCE last week instead of BEFORE last week.

One other feature is the ability to specify that only the Job Repository entries should be cleaned up. Liberty Batch is also keeping joblogs for each job (that we'll talk about later on) and purge can clean these up too. But if you have some other mechanism managing those log files, then you can tell purge not to bother hunting them down and just tidy up the repository.

REST Interface page and pagesize Parameters

All of the GET services that can return more than one result, as well as the DELETE service to purge jobs, support the *page* and *pagesize* parameters. What do those do?

The idea is that some services, such as GET a list of all jobs, might return a really really large result set. There could be tens of thousands of jobs in the Job Repository and you might not really want all of the information about all of those jobs coming back in response to your request.

The *pagesize* parameter is used to limit the number of records returned. The default is fifty. Well, if we have a page size, that means the results are broken up into pages of that size. And so you can use the *page* parameter to indicate what page you want.

Suppose you GET a list of jobs matching some criteria and there are two hundred such jobs. By default the GET request will return the first fifty of those (*pagesize=50* and *page=0* being the defaults). To get the next fifty, you need to specify *page=1* and so forth to get the remaining three pages of data.

Of course, you can also change the page size (to 200 in this case) and get all the results in one response. If you don't know how big the result set is and you are sure you want it all no matter how big it is, just set the page size to something really big.

There is, of course, a small problem with this approach. The state of the Job Repository may be changing between requests. The result of a request isn't actually kept around between calls since we have no way of knowing if you'll be back for the rest of it. If you ask for the first 50 of a result that contains 200 records, then come back and ask for the next 50, there is no guarantee that nothing has changed in between. A result in the first set could now be part of the second 50 records (possibly if somebody purged some jobs that were in the first 50).

Which brings us to the really weird case. Suppose you are using DELETE to purge jobs. Suppose you are trying to purge those 200 jobs we found earlier. If you purge with *pagesize=50* and *page=0* (defaults remember) then the first 50 jobs matching the purge criteria will be deleted. But that leaves only 150 jobs remaining, and if you treat it like you did search and move on to *page=1* you will purge the second 50 results, but those are really the third 50 results from the original query as the new result set is only 150 jobs. Once that completes there are only 100 jobs left and a matching purge with *page=3* will delete nothing as there aren't any jobs on that 'page' of the results anymore. But there are still 100 jobs left. The trick here is to just keep deleting *page=0* until there is nothing left. Or do a GET and page through the results (which hopefully aren't changing too much underneath you) to get a count and then set the DELETE *pagesize* to that amount.

Hopefully that was clear…. Our song this week is obviously "Turn the Page" by Bob Seger.

Application Packaging and In-Line JSL with the REST API

We haven't talked about how a batch application is packaged. You can package them as web applications (.war files) or as enterprise applications (.ear files) and deploy them into a server just like any other web or enterprise application.

The archive file contains all the Java .class files needed for the batch application (ok, it can be more complicated, but we'll go with that for our purposes here). But the specification also says that the JSL is contained inside the same archive file in a folder called batch-jobs under the META-INF folder.

Well that's pretty nifty. The JSL that defines what the flow through the various bits of Java code that make up the batch job will be lives inside the very same archive file that contains all that application code. For DevOps stuff where you're trying to manage keeping all the bits of an application together, moving it from dev to test to prod, having it all in one archive makes that pretty easy.

But if you're from a z/OS background you might be used to thinking of JCL, application source code, and application binaries as very separate things. Maybe you library the source code somewhere and have a separate process for managing the binaries and the JCL. Keeping this all together just isn't how you do things. That's ok..

One of the features of the Liberty Batch REST interface is the ability to provide the JSL along with the request to start the job. In a later post we'll get to how you'd integrate submitting Liberty Batch jobs in an Enterprise Batch environment, but for now just know that this makes it possible to have the JSL as instream data inside JCL.

Essentially your Enterprise Scheduler will run JCL, just like it does today, and that JCL executes something (we'll get there pretty soon) that drives the REST interface to submit the Java Batch job and the JSL is right there in a "DD *" entry in your JCL (or off in some dataset a JCL DD card points to).

But this applies to uses in non-z/OS environments too. Anyplace that you are using the REST interface to submit a job you can choose to have whatever client is driving the interface managing the JSL and supply it as part of the job-start request..

This capability is all part of the WebSphere Liberty implementation of JSR-352.

batchManager Command Line Interface

Here's a challenge – think of a clever and witty title involving the words "Command Line Interface". I spent a while trying to find some song lyrics that went with it, but I gave up. Perhaps there's an Italian opera that uses it and I don't realize it.

We've been talking about the REST interface that you can use to interact with the batch support inside a Liberty server. That's all well and good if you've got some application code that needs to reach into a server and submit a job or find out what happened to a job submitted earlier. But what if you have a scripting environment or some JCL running on z/OS? How would that use a REST interface to manage Liberty Java Batch jobs?

Well, you could write a little client program that drove the REST interface. It isn't terribly difficult to do, which is why we did it for you. Liberty ships a little program in the 'bin' folder called `batchManager`. Underneath it is a Java program that parses a bunch of input parameters and constructs a proper HTTPS/REST request to submit a job or whatever other operation you need to do.

Of necessity the parameter options can get a bit complicated because there is a lot you can do. But if you just work with the - - help (two dashes, then 'help') option it will tell you all your choices.

Having a lot of complicated options means you won't likely actually issue the command yourself at a prompt. The safer/easier thing to do is construct a small script that executes the CLI (that's Command Line Interface when I'm too lazy to type it out) with whatever parameters you need.

You'll likely have a set of scripts that you use to submit jobs. But you might also have a script that you use to purge older jobs. Chron or some scheduling mechanism could kick that off and jobs older than whatever your standard retention period is would get purged automatically. The CLI has all the fancy search options we talked about under the REST interface.

So, find your bin directory and play around with the help and see all the different things you can do with batch from a script.

WebSphere Developer Tools and Batch REST

We've been talking about the batch REST API and ways to exploit it, so that seemed like a good segue into talking about WebSphere Developer Tools. WDT (as it is known to its friends) is a plugin you can install into Eclipse. It provides a lot of tooling and wizards to help you interact with Liberty. It can guide you through creating a complex server.xml configuration for Liberty, and it puts all the close brackets and quotation marks in the right places (which is better than I do).

As part of that, WDT also understands Java Batch. We'll save most of that for another day, but for now I wanted to talk about the interactions with the batch REST interface. If you create a Liberty server inside Eclipse, then WDT knows about it. And you can create a Run Configuration to run a Java Batch job.

If you have installed an application into the server, then WDT knows about it and knows about the JSL files that are in that application. The Java Batch Run Configuration template allows you to pick an application and JSL from selection lists. Then just specify a valid userid/password for the target server (to pass authentication/authorization checks) and you're ready to go.

But wait, there's more! WDT has read through the JSL and spotted any job parameters that are mentioned. You can add a parameter to be supplied over the REST interface (along with a value for the parameter, of course). Since WDT knows the parameter names, it can provide a selection list which prevents you from spelling the parameter name worng and then wondering why it didn't do what you wanted.

WDT can also query a list of jobs known to the Job Repository (remember that can be in-memory or in a database, depending on configuration – so jobs might disappear if you restart the server). And from the job list you can select a job and WDT will use the REST interface to fetch the job log. Of course, you can just go look at the log in the file system on your laptop where you're running all this, but it can be handy to see it in Eclipse along with the application code and the JSL.

So, for Liberty servers running inside Eclipse, WDT provides a nice way to submit and observe jobs running in that server, all right there in your development environment.

SideBar: Using WDT to Develop Batch Applications

We’re going to step aside from talking about the REST interface and ways to exploit it to follow up on the WebSphere Developer Tools (WDT) and how they can help you create Java Batch applications.

The batch support in WDT allows you to create a Batch Project and inside of that you can create Batch Jobs. Creating a job creates a JSL file in the batch-jobs folder. The cool thing about using WDT to create the job is that it knows all the XML syntax for JSL. You can use the wizard to create steps, chunks, properties, listeners, etc.

This is especially awesome if you are using substitution with job parameters, properties, etc. You might recall the rather complicated syntax with all those quotes and brackets and so forth. WDT understands that all and has a little panel where you can explain what you want, and it will generate the correct syntax.

As you define the job, you’ll start specifying various Java artifacts that you’ll need to use. Your JSL will specify a batchlet, or an `ItemReader` or a Listener. WDT knows what the interfaces are for all of the artifacts and can help you by creating a skeleton implementation for you to fill in. It even knows all the various listener types. If you add a listener to the JSL, it will ask you which type you want and set up the proper skeleton.

I don’t want to go through specific directions to create and run a job here (there are a few writeups/labs about doing that in DeveloperWorks and IBM Techdocs). I just wanted to make you aware the tooling existed. I use it all the time developing samples and tests for various things and it certainly prevents me from making a lot of simple, annoying mistakes.

The Admin Center Batch Tool – Part 1

WebSphere Liberty servers can be managed using the Admin Center which you can enable as a feature in the server. I won't try to go into all the things you can do with it here, but the various features are grouped into "tools" which you can enable. One such tool is the Batch tool.

The Admin Center Batch tool is really an application that runs in your browser and uses the Liberty Batch REST API under the covers to provide function. What function is that? Well, it is essentially a web browser interface to look at (and manipulate) the Job Repository.

Using the REST interface under the covers the Admin Center Batch tool can fetch a list of jobs and display that list to you. Using the search capabilities we've discussed earlier, the tool can craft REST requests that filter or scope the resulting list to jobs that you are interested in. For example, you can see all the jobs submitted by a particular user or run after (or before) a certain date.

Once you have found the job you are interested in, you can select it and drill down to get more information. Again, the Batch tool is just using the REST API to fetch the details kept in the Job Repository about the job, but this allows you to see all the steps in the job and their status. Wondering how the third step in the job that ran yesterday turned out? The Batch tool lets you find that job and explore the step to see what happened.

We haven't talked about Job Logs yet (we'll get there) but the Batch tool can also fetch the log for a job and let you browse through it.

If you're familiar with SDSF on z/OS then the Batch tool is providing similar function. It is an interface to let you see everything that has gone on and explore it in detail.

Well, ok, maybe not everything that has gone on. You will need to logon to allow the Batch tool to use the REST interface and your identity could limit what jobs you can see. Security is another topic we'll get to eventually, but, as we've said before, the REST interface is secured so you can't access jobs you aren't authorized to and the Admin Center's use of the REST interface inherits that security.

That's all 'read' operations from the Admin Center. Next time we'll look at 'write' operations..

The Admin Center Batch Tool – Part 2

Last time we talked about using the Liberty Admin Center's Batch Tool to browse through a list of jobs and dig into what went on in those jobs. We can look at the results of each step in a job and even call up the joblog. There are a lot of search and filter capabilities to make it easier to find the job you're looking for.

All those are basically "read" operations. Are there any "write" operations you can perform with the Admin Center? Well sure. As we've noted before, the Admin Center is really just a pretty client that uses the batch REST interface. And the Admin Center has taken advantage of a few different "write" operations that act on jobs in the Job Repository.

The first of those is "stop". As we've noted in our discussions about the JSR-352 specification there is an option to try to stop a running job via the `JobOperator` interface (and via the Liberty Batch REST interface). This will stop a job in a chunk step if/when it progresses through a read/process cycle and the batch container code can check to see if the job is stopped. For a batchlet it will drive the `stop` method which informs the batchlet about the stop request and it is up to the batchlet to try to stop itself. The Admin Center allows you to select a running job and drive a stop request against it.

If a job has failed and you want to restart it, you can also do that through the Admin Center Batch Tool. Just select the job and then select the Restart operation. You'll be given an opportunity to specify new Job Parameters to be used when the job is restarted. In a production environment a failed job might get restarted by automation, but you'll want to test that out before production and using the Admin Center to drive the restart is a handy way to do it.

Finally, you will eventually end up with jobs in the Job Repository that you just don't care about anymore and you want to purge them. It is likely in a production environment that will be done with some automation driving a script that uses the command line interface. But there might be cases where you just need to purge a job or two and, again, the Admin Center Batch Tool is a handy way to do that.

Enterprise Scheduler Integration

If you're going to run production batch applications, you're going to need something to coordinate and manage them. There are a bunch of different ways to do this including several software packages you can use.

The question for today is: How would you integrate Java Batch applications running inside a Liberty server with whatever your existing Enterprise Scheduler is?

Enterprise Schedulers run things. Probably a script or something like one. On z/OS they generally submit JCL.

Java Batch applications are available to run inside a Liberty server. How do we connect the two?

Well, one way would be to teach your Enterprise Scheduler about the Batch REST interface. When it was time to run a Java Batch job, the scheduler could just use the REST interface to start it. Later we'll look at an Enterprise Scheduler that has done exactly this.

You could also have the scheduler run some program that drove a request into a server and the dispatched thing (a servlet, an EJB, etc.) would use the `JobOperator` interface to start the job.

Another approach would be to have the scheduler run a script that executes the `batchManager` Command Line Interface. It uses the REST interface under the covers to submit the job. The CLI is capable of waiting for the job to complete too. That means the script won't end until the job ends which means the scheduler just runs something and when it is done, the job is done.

One more approach I'll just mention briefly is the ability to simply have the scheduler start up a Liberty server where application initialization starts the job running. You can find an example of this in the Tweet Analyzer sample in github under `WASdev/sample.batch.tweetanalyzer`.

As we noted under the discussion of the CLI, the tricky part of this isn't always getting the job started but knowing when it is finished (for good or bad). And if the job fails and you want to restart it, how do you automate that with an Enterprise Scheduler?

Enterprise Scheduler Integration with Scripts

One approach to integrating Liberty Java Batch applications with an Enterprise Scheduler is to have the scheduler execute a script that, in turn, uses the `batchManager` (or on z/OS `batchManagerZos`) Command Line Interface to submit a job.

That seems simple enough, but there are a few things you'll want to consider. First of all, you should be sure to specify that you want the CLI to wait for the job to complete. Otherwise the CLI will submit the job and return and the script will end. To the Enterprise Scheduler that looks like the job finished and it will be free to go do whatever comes next when, in fact, the job is still running over in Liberty somewhere.

You also want to consider whether or not you want the job output in the output from the script. We haven't talked about joblog processing yet, but for now just know that it is possible to get the output from the job back from the invocation of the CLI, if you want it. If you do, then it will end up in the output from the script (the CLI basically prints whatever log it gets back). That will make the job output available as part of the script output and thus available to the Enterprise Scheduler, if that helps. Otherwise it still exists and is accessible later but having it where the scheduler expects it to be might make things simpler for operations.

Another consideration is restart processing. If the job fails, you might want your scheduler to know about that and automatically restart the job. How does that work? The CLI has a parameter that allows you to specify a restart file. The CLI will write information (the job instance ID) into that file when it initially submits the job. On the other hand, if the CLI finds a job instance ID in the file already, it will assume you want that job restarted and will perform a restart operation instead.

That means the name of that restart file needs to be associated with this particular job instance as known by the Enterprise Scheduler. For example, if the scheduler runs a particular job every Tuesday, then you'd want the date of THIS Tuesday to be part of the file name. If the job fails and needs to be restarted, you want the scheduler to use the file with that date and not a new file, so the failed job gets restarted. Successful completion of a job will clear out the restart file.

On z/OS there is some weirdness with return code handling from `BPXBATCH` (the utility you'd likely use to launch a script). I won't go into the details here, but you need to take some special care.

There are samples of using both `batchManager` and `batchManagerZos` from scripts in GitHub in the `WASdev/sample.batch.misc` project.

The IBM Workload Scheduler

Last time we talked about how you can use a script running the `batchManager` (or `batchManagerZos`) Command Line Interface with an Enterprise Scheduler to manage Liberty Java Batch job execution. This time I want to cover what is (at this writing) the special case that is the IBM Workload Scheduler (IWS – formerly known as Tivoli Work-load Scheduler (TWS)).

Of course, you can use IWS like any other job scheduler and have it execute a script that functions like we talked about last time. But IWS went one step further and created a plugin that uses the Liberty Batch REST interface.

The plugin allows you to define a Java Batch job as part of an IWS workflow. When you define the Java Batch job you have to provide the host/port of the Liberty server that should be the target of the REST request. Then you can provide the same parameters you provide to the CLI such as application name and JSL file name. You can also pro-vide job parameters to be passed on the start request.

If you go back a few weeks, you'll find our discussion about supplying the JSL "inline" instead of using JSL packaged with the application. The IWS Java Batch plugin also al-lows you to specify JSL in the panel where you define the job. This lets you keep the JSL in with your IWS-managed workflow, if that makes sense for your environment.

IWS can also make use of the REST interface to monitor the status of the job and pro-vide access to information about the job after it is completed (just like WDT does for batch jobs running inside Liberty-in-Eclipse).

And finally, you can use IWS to fetch the joblog for the job and view it in the IWS con-sole.

If your Enterprise Scheduler of choice is IBM Workload Scheduler, the JSR 352 Java Batch plugin provides close integration with Liberty Java Batch. But, of course, it does so using the documented external batch REST interface, so it isn't doing anything any other scheduler couldn't also do.

What About the Joblog?

Batch jobs write messages. Where do those messages go? In a traditional batch environment, there will be some sort of log produced by the job that you can look at to see how things are going, maybe debug a problem. What happens to those messages for a Java Batch job running inside Liberty?

Well, like any message written by an application running inside Liberty, they end up in the server output. But that's not very helpful. More than one job can be running concurrently inside a server at the same time and their messages will get all mixed up together in the server log. It would be nice if each job had its own log somewhere.

And so they do. The Liberty Batch implementation creates a folder called `joblogs` in the server's `logs` folder. A directory structure underneath that separates the output from jobs by application name, then by date of execution, and then by instance and execution identifiers.

At first, that seems like it would be enough. Each job execution then gets its own separate joblog file and they are managed in a folder hierarchy that makes it easy to find the one you want. But, of course, nothing is every as easy as it seems like it should be.

Remember that the JSR-352 specification supports concurrent execution models with a partitioned step and with a split/flow. When the application has multiple parts running concurrently, you don't want them all writing into the same joblog file. That's going to make a mess of things.

For that reason, more directories are created under the execution for a split and the flows within it and each one gets its own job log. Similarly, partitioned steps get separate log files for each partition.

Which means that while a nice simple "Hello, World!" batch job will have a nice simple job log file, for a complex job with nested Splits and partitioned steps you'll have a pile of different log files for each thread of execution – in addition to the main log for the main thread of the job itself.

You can also ask the Liberty Batch implementation to break the log into parts if you want to keep the individual log files from getting too big (whatever "too big" means to you).

Next time we'll look at fetching the joblog(s) using the REST interface.

Accessing Joblogs Using the REST Interface

Getting a joblog through a REST interface sounds like it should be a pretty simple thing. You just need the right REST URL to ask for a joblog and some parameters to tell the REST service which job you care about.

As we saw last time, it isn't that simple. A job log can easily consist of a whole set of log files written by the main job thread itself as well as logs written by various pieces of the job that run concurrent to the main job thread (partitions, split/flows). When you ask the REST interface for the log for a particular job, which of these files do you want?

The easiest way to get the job log you want is to first know the execution identifier of the job execution for the job you care about. Use the `jobexecutions` REST interface specifying that id in that path followed by 'joblogs' (see the Liberty Knowledge Center for syntax specifics). That tells the REST service that you want the entire log (all the parts) for the job execution you specified.

Under the covers the REST interface will find all the various files that make up the log and concatenate them together and return it as a single text response. Looking at the merged log you can easily tell where the different log files that went into it start and stop. That's probably the best thing to do if you are retrieving it to display it somewhere.

But if you just want it back to archive it someplace, it might be handy to zip it up. To help with that, the REST interface also supports specifying type=zip (vs. type=text which is the default). That will return a binary zip file consisting of the entire log file tree for the job execution you specified. That's less handy for immediate use, but if you are just archiving it might save some bandwidth carrying large logs around.

That's all great, but it is possible you know exactly which part of the log you care about. Maybe it is just the flow called "flow2" that is part of a split called "MainSplit" in the main job thread. Knowing that it is also possible to specify a part keyword that gives the specific path to the part of the log you care about within the particular job execution and just retrieve that one log part.

Of course, this is all only interesting if you are working with the REST interface directly. Remember that the Command Line Interface (`batchManager`), the Eclipse plugin (WDT), and the Admin Center Batch tool all use the REST interface for you and make the job logs readily available.

Batch Events Overview

You have a Java Batch job running inside a Liberty server, but what it is doing? You can use the REST interface to find out what step it is executing and fetch the job log. But what if you want to log job activity somewhere? What if you want to react when something important happens to the job (like a step fails)? What if you have an executive who always wants to know about the status of some important job? How can you monitor the job?

Batch Events! The Liberty implementation of JSR-352 has the ability to publish messages (events) to a messaging topic tree whenever interesting things happen to the job. In subsequent posts we'll look at some of the different types of events that generate messages and what you might do with them.

At a high level, there is a topic tree consisting of different events in the life of the job (from the submission of the job through the completion) and downward in granularity through steps and even checkpoints for a chunk step.

Messages about all the jobs running in an environment are published into the same topic tree. You can configure a different topic root for different batch environments (e.g. prod and test, or maybe separate by lines of business).

Messages contain information about the job or step, similar to what you would get from the REST interface if asking about the status of the job or step.

More importantly, all the messages have two (sometimes three) message properties. The two you always get are the job instance and execution identifiers. If the message relates to a step, you get the step execution id also. The job instance and execution identifiers allow you to separate messages for different jobs from each other.

If you know the identifier for the job you are interested in (because you submitted it) then you can subscribe and filter to just receive messages from particular topics (completed, failed, etc) about the job you care about.

All this allows you to build a central monitor watching all the jobs or build a particular monitor that watches for events within a specific job. You might do this with Message Driven Beans running in some Liberty server that is subscribed to the topic tree.

In the coming weeks, we'll traverse the topic tree and have a look at all the different event types.

Job Instance Events

Last time we talked about the ability of the Liberty JSR-352 support to publish messages (events) into a topic tree at certain points in the lifecycle of a batch job. This week we'll take a look at what events are available relating to a job instance.

Before we get to that, let's remind ourselves what a job instance actually is (as compared to a job execution). It all has to do with restarting failed or stopped jobs. When you submit a job, you create a job instance. When the job starts running, that's a job execution under that job instance. If the execution fails and you want to restart the job, you get a new execution under that same instance. Contrast that to just submitting the job again which creates a whole new instance (and execution).

Got it? Great! Since we are talking about submitting jobs and creating instances, this would be a good point to talk about the events relating to that. Any time a new job gets submitted a message will be published into the `batch/jobs/instance/submitted` topic. As we mentioned last time, that's the relative location in the topic tree based off wherever the batch topic root is configured to go.

Alright, so that one event would allow you to create a monitor that was notified anytime a new job is submitted. What other events get published? Well, when the job starts running you'll get a message in the `dispatched` topic (under `batch/jobs/instance` also, as are all the topics we'll talk about today). Knowing when dispatch starts merged with the submit allows a monitor to know what is waiting to execute (and the message content tells you what those jobs are).

As jobs finish messages are published into one of several topics: `completed`, `stopped`, or `failed`. All of those indicate the job ended, but the specific topic the message shows up under tells you how. Note that a stopped job will have posted a message to the `stopping` topic when the stop was issued.

Again, all this allows a monitor to know what jobs are running and what the state of completed jobs is, as well as identify jobs that have been stopped but haven't yet actually ended. Of course, you can get all this information from the Admin Center or by using the REST interface to query the state of things. Being notified about events allows a monitor to just sit back and observe without having to actively poll. And a monitor could then generate its own alerts based on whatever criteria makes sense.

Ok, there are three more job instance related events. Two of them have to do with multi-server configurations that we haven't talked about yet: `jms_queued` and `jms_consumed`. Messages are published in these topics when jobs are queued or taken from the queue in a multi-server configuration.

The final job instance event is `purge`. A message is published there when a purge operation has cleaned up the job from the Job Repository. If you are managing an archive of job logs somewhere, you might trigger off a job-purge message to move the job log from a local archive to a tape library (or get rid of it entirely).

Job Execution Events

Last time we talked about the nodes in the batch topic tree where messages (events) get published at significant points in the lifecycle of a job instance. This time we want to take a different branch and look at job execution events. Those are under `batch/jobs/execution` instead of `batch/jobs/instance`.

Just quickly, a job instance is created when a job is submitted. A job execution is created when the job is run or restarted. One instance can have one or more executions.

Which means that a job execution has some of the same events (topics) as an instance. Those are the ones that cover when the job ends. There are events for the execution being completed, stopped, and failed. These will happen in addition to the similar events for the job instance.

A job execution does not have stopping or purged events because those operations occur at the job instance level.

A job execution doesn't have the submitted or dispatched events (or the JMS-related events either). Instead we have `execution/starting` and `execution/restarting` events that tell you WHY this execution is here. You only get an `instance/submitted` event for the original creation of the job. And the `instance/dispatched` event doesn't tell you why, you need the `execution/starting` or `execution/restarting` events to tell you that.

So what? Well, again consider that you might be building some sort of monitoring that would subscribe to these various events and try to show what is going on. You would need this information in order to show (in some fashion) a job that was restarted and running vs. a job that was on its first try vs. a job that had failed and hadn't been restarted yet. The distinction might make a lot of difference.

If you look closely at the topic tree, you'll notice there is one more node directly under `batch/jobs/execution` that I skipped. I'm saving that for next time..

Job Log Events

There's one more topic in the `batch/jobs/instance` topic tree that we haven't talked about: `jobLogPart`. A message gets published to this topic whenever a job log part is completed. When does that happen?

Recall that the main thread of the job writes messages to a log file. You can configure that log to "fill up" after a number of messages (default is 1000) or after some time period has expired. When the log is "full" the file is closed and a new one created for messages from the job going forward. As part of closing the old file a message is published to the `jobLogPart` topic.

That message has the job instance and execution identifiers as properties. That enables a subscriber to filter on messages relating to a specific job. The content of the message is the job log contents itself.

Remember also that if the job has partitions or uses the split/flow construct then several threads may be executing for the job besides the main thread. Each of those threads writes into its own separate log file. When those logs are "full" or the thread ends (the partition or the flow) then the log file is closed and the content published to the `jobLogPart` topic.

If you have a central location where you want to keep all the logs from all the jobs, then rather than have some tooling use the REST interface to go harvest the job logs, you could establish a subscriber to the `jobLogPart` topic and be handed the job log contents as they fill or complete. A Message Driven Bean (MDB) might be ideal for this purpose.

If you'd like to feed your batch job logs to some log analysis service, subscribing to this topic would allow that service easy access to all the log contents.

Just remember that log parts messages are published in the order the log files fill or things complete. If you have four partitions running concurrently producing enough output to cause the log files to reach the configured maximum number of records, then messages will be produced for the four partitions as they fill and move to new files. The messages for one partition can interleave with messages from other partitions, so be careful.

Next time we'll move in one layer and look at step-related events.

Step Events

We started sort of from the outside with Job Instance events and then moved inward to Job Execution events. Now we'll go inside the job to events generated at the Step level.

As each step begins a message is published to the topic for started steps. As before, the message has properties identifying the job instance and execution. But since we are at the step level there is also a step execution identifier included as a property. This isn't something you can know ahead of time but having seen the property for the start of the step it would make it easier to find other step-related messages.

The content of the message is the same JSON string returned by the REST interface when asking for information about the step (using the step execution id).

As with the job itself, a message is published to an appropriate topic as the step ends. Which topic depends how the step ends. There is a `completed` topic for steps that end normally. This may be the final step in the job or flow might transition from here to another step.

If things didn't go well, a message is published to the `failed` topic for steps. A failed step will cause the job to end and might be a handy thing for a monitor to recognize and raise an alert about.

If the job was stopped, either because an operator stopped it or because the job internally stopped itself (remember that end-of-step flow transitions can include stopping the job) then a message is published to the step `stopped` topic.

The final step-related topic is `checkpoint`. Every time a chunk step takes a checkpoint, a message is published to this topic. Some chunk steps might have a fixed (or at least predictable) number of checkpoints. Monitoring the checkpoint events would then give you a count of checkpoints against the known total and the ability to present a progress bar for the step in a monitor.

However, this might not always be possible. Of course, some jobs might have a varying number of checkpoints depending on how chunking is done (time based vs. item count based will make it hard to predict). Error handling within the job can influence the checkpoint count also. If retry-rollback exceptions are defined and one is thrown the step will rollback to the last checkpoint and then retry, checkpointing one record at a time, until the problem record is passed. This process would substantially throw off the number of checkpoints seen from what was expected. Still, it is a neat idea in cases where it will work.

Next time we move further inside the step to partitioned steps.

Partitioned Step Events

If you are monitoring a job that has a partitioned step, things are going to get pretty exciting. Once the `started` message is published for the step, the process of spinning off the partitions will begin.

Each partition will publish its own message to the `started` topic under step partitions. You might know how many partitions to expect for a given job if the partitioning is hard-coded in the JSL (or done programmatically into a fixed number of partitions with varying work for each partition). But nothing says that all the partitions start at once. The application can limit the number of partitions that can run concurrently, and, of course, availability of threads and processor resources to run them can restrict concurrency.

As the partitions run, they may publish messages for checkpoints as they take them. But once all the fun is over, each partition will publish a message indicating how it finished.

As usual, when things work well a message is published into the partition `completed` topic. Of course, that doesn't necessarily mean everything worked as you'd hoped, just that no exceptions were thrown out of the partition. Check the exit status in the message content to see what the application actually thought about how things went.

If an exception does get thrown out of the partition and not handled through skip or retry processing, then the partition failed, and a message will be published to the partition `failed` topic.

As with the step messages, if the job is stopped while a partition is running then the partitions noticing the change in status will publish a message to the `stopped` topic for partitions.

That's all there is for partitioned step events. For jobs where the partitioning is at least somewhat predictable it should enable a monitor to display progress and status of the various partitions.

Split/Flow Events

The last in our series about batch events (finally!). This time we're going to look at the event messages published around the processing of a split/flow.

We talked about those quite a while ago, so let's review. By wrapping `<flow>` elements around one or more `<step>` elements in a job we can define a flow. The flow has a name (just like a step or any other element in the job).

If we have one or more flows together, we can wrap `<split>` elements around the flows and create a split/flow (of course!). When processing for the job proceeds to the split element (which also has its own identifying name) processing on the thread of the job encountering the split stops and processing for each flow within the split is spun off to other threads.

When all the steps within a flow are completed, meaning a step with no transition to another step was encountered, then the flow is done. When all the flows are done, the split is complete.

What batch event messages get produced for all of this? Well, nothing happens from the split itself. The thread encountering the split just stops until all the flows are done. However, as each flow begins a message is published to the split `started` topic. You'll get one started message for each flow within the split. As with partitions, these might not all happen at once depending on availability of threads to run the flows and processor resources to run the threads.

Once the flow is running, it produces step-related events just like any other step outside a flow. Remember that a flow can contain regular and partitioned steps. And, brace yourself, a flow can contain another split that contains more flows. So this can get pretty complicated, if you let it.

Once all the steps in a flow are complete, the flow ends, and a message is published to the split `ended` topic to let you know that flow is done.

Without knowing the JSL you have no way of knowing how many flows are within a split so you can't really show the status of all the flows for sure. Just the ones you've seen started and ended.

That's it for batch event messages. Next time we're on to a whole new topic…

Introducing Batch Server Topology Considerations

This post will kick off a series discussing various topology considerations when running Java Batch applications in Liberty. At first glance you might wonder what kind of topology you can even have. All our discussions so far have involved using `JobOperator` or the REST interface to submit a job to a Liberty server that runs the job. There's not a lot of topology involved.

But that server keeps track of things in a Job Repository that is really just a set of tables in a database. Could multiple servers share those tables? What consequences would that have?

To make things more interesting, the WebSphere Liberty implementation supports a multi-server configuration with some servers acting as 'dispatchers' and others acting as 'executors' which actually run the jobs. The work is passed from a dispatcher to executor via a messaging queue. This capability introduces quite a few interesting topology possibilities.

And, of course, you could have several different Job Repositories and several sets of servers acting as dispatchers and executors and multiple different queues. You could create quite a mess.

Adding one more twist to this is the batch events topic root. Remember that a server can be configured to publish 'event' messages to a topic tree. The root for that topic tree (and for that matter, the messaging system containing the tree) is configured as part of the server configuration. You could have multiple topic tree roots being used by different servers if you wanted to separate them. Would you? Would that separation line up with sets of dispatchers and executors or users of particular Job Repositories? Again, a great opportunity to make a big confusing mess of things.

We'll get into all this in more detail, but, as a quick rule, remember that job identifiers (instance and execution IDs) are unique with a Job Repository. Those identifiers show up in messages between dispatchers and executors and in event messages published to the topic tree. That suggests some alignment between users of a particular set of Job Repository tables and users of these other artifacts. You don't want a server using one repository to get a message from a server using a different repository because the identifiers are referring to different actual jobs.

That should be enough to get you worried. Over the next few weeks we'll try to sift through how all this works..

Sharing a Job Repository

The Job Repository is really just a set of tables living in a database. Multiple Liberty servers can easily be configured to use the same set of tables. They can just as easily be configured to use separate sets of tables. When should servers share a repository (if ever) and when should they stay separate?

The key, as we mentioned last time, is the numbers (identifiers) that represent the jobs. Every time a new job is submitted it receives an instance id and an execution id. Those identifiers are unique within the Job Repository. So, servers sharing a repository will get unique identifiers from one set. Servers using a different repository can get the same identifier values, but they obviously represent different jobs.

Thus, at some level, when someone says to you, “I need the output from job 75348.” will you be able to figure out which actual job that is? Is it obvious from the context of the question which repository they are using and thus which actual job that is?

Consider it from another direction. If you make a REST request to a server asking for a list of jobs meeting some criteria, that list is generated from the Job Repository being used by that server. If that repository is shared by other servers, the returned list might include jobs run in those other servers. Does that make sense to you?

Probably development, test, and production should have separate repositories. The earlier stages might have quite a few different repositories (each developer might have their own or be using an in-memory repository). But in production, is one enough? Should you partition them by Line of Business or some other organization boundary? Or just have one big production repository that is used by everything?

Most factors generally encourage you to use a single repository for a production environment, although there are certainly reasons to separate them. I’ll just bring up one here..

Remember the repository doesn’t just keep track of the jobs that have run and their final status. Lots of in-flight information about running jobs is kept in the repository. For a chunk step, every checkpoint requires updating checkpoint information in the repository. If the database containing the repository tables is physically distant from the server running the job the latency of pushing that update to a remote database at EVERY checkpoint (of thousands, maybe millions) could significantly impact the elapsed time to execute the job.

Best practice? Whatever works best for you : -)

The Multi-Server Batch Configuration – Dispatchers

All the discussion we've had so far starts with contacting a server to run a job and the job runs right there in the server you contacted. That means the application (.war file or .ear file) that contains the application has to be installed in that server. Which means, if you have a lot of applications installed in a lot of different servers, that you need to know which server to contact depending on which job you want to run.

Wouldn't it be nice to just have a single point of contact you could tell "Please run this job" and it would figure out which server was configured to run that job and routed your request there? The "Dispatcher" in a multi-server Liberty Batch configuration was designed to solve this problem.

You configure a dispatcher by adding a `batchJmsDispatcher` element to a Liberty server configuration. This element provides references to connection factory and queue configuration leading to a defined queue in a messaging engine. The messaging engine can be the one included in Liberty or the IBM MQ product.

The presence of this configuration tells the server that any attempt to submit a job through the REST interface doesn't really want the job run in this server, but instead in an appropriate server configured to run it. The dispatcher will go ahead and create the entries in the Job Repository for the job (so it has an instance ID) and then put a message representing the job into the configured queue. Next time we'll talk about configuring the servers to process those messages.

Alright, so now we have a single server that any REST client can contact to submit a job and we'll trust that it gets to the right place to run (it works…we'll get there). But that phrase "single server" raises concerns. What if the dispatcher is down?

With the configuration we started with (contacting the right server for the job directly) a dead server only meant that jobs for applications living in that server couldn't be run. And you might have set up several servers hosting the same applications. But now, with one dispatcher server front-ending everything, if that server is down the whole batch capability is broken. What to do?

Create more! There's nothing magical connecting a particular dispatcher to the executors we'll talk about. All the dispatchers and executors need to be sharing the same Job Repository tables, so they are on the same 'page' about what a particular job id value means. But you can create as many dispatchers as you need and REST clients can choose freely among them, just like any other REST service hosted on replicated servers. All the job state is kept in the Job Repository so there is no affinity established to a dispatcher.

But the dispatcher is useless without executors… next time!

The Multi-Server Batch Configuration – Executors

Last time we talked about configuring a WebSphere Liberty server to act as a batch dispatcher. The results of a job being submitted via the REST interface to the dispatcher are entries in the Job Repository about the job and a message in a queue that represents the job.

This time we'll look at what happens to that message to get the job running. To get things moving we need another server that includes the `batchJmsExecutor` configuration element. This element includes attributes that point to the configuration for an activation specification and a queue. This configuration tells the batch executor code how to find the queue where the dispatcher is placing messages.

When a message turns up, an executor will pick up the message and process it. This is very similar to how Message Driven Beans (MDBs) are processed, although with some special 'batch' code doing the message handling.

The batch code will look at the message contents (which includes the job identifier) and find the information in the Job Repository for this job that was placed there by the dispatcher. This is why the dispatchers and executors using the same queue need to be sharing the same Job Repository.

Once it knows what to do, the batch code will launch the job. The job will run (hopefully to a successful completion) and make updates in the Job Repository as it progresses. These updates enable REST clients to make requests asking about status of the job to a dispatcher and get the right answers. The clients submitted the job, but don't really need to know which server wound up running the job. The dispatcher can give them access to job status without needing to know.

Of course, you can create more than one executor to process messages from the job queue. You might want to have different applications hosted in different servers, so there needs to be a way to make sure the right messages get to the right servers.

That's our topic for next time!

The Multi-Server Batch Configuration – Job Routing

We've been talking about how to solve the problem of needing to know which batch applications are installed in which servers, so you know which host/port to use to direct the REST request to submit a job. So far, we've shown how configuring a WebSphere Liberty server as a dispatcher (or several servers as dispatchers) creates a front-end for the REST requests. The dispatcher(s) then put a message on a queue that is picked up by an executor server which runs the job. But we still haven't seen how jobs end up in the right servers.

The basic configuration needed to establish a server as an executor just points to the right queue and defines an activation specification needed to get messages when they show up. But if we go just a bit past that we can solve the last part of our problem. Part of configuring an activation specification can be the definition of a message selector string. This somewhat SQL-WHERE-clause-like string defines what messages should be handled by this server (instead of just handling any message that shows up).

Message selectors allow you to select messages based on the attributes of the message. This can include properties assigned to the message. For example, if you know some messages on a queue will have a property named CLASS and you want to process only messages where that property is defined and has a value of "ABC" you can specify a message selector string of CLASS='ABC'. You can AND and OR together different specifications and some other fancier stuff to get exactly the messages you want.

So what? Well, it turns out that when the dispatcher puts the message on the queue representing the job, it sets some properties on that message that you can use in a message selector string. The first, and probably most important, is the application name. The name of the application (basically the .war name) is a required parameter to submit a job using the REST interface. The dispatcher creates a message property called `com_ibm_ws_batch_applicationName` and sets it to the specified application name. Which means you can use that application name in the server configuration in the message selector to make sure the server only processes messages that will run batch jobs using applications that are deployed in this server.

Problem solved! But wait, there's more! The dispatcher also takes any job parameters that are specified as part of submitting the job and turns them into properties on the message. Thus, if you submit a job with a job parameter of JOBCLASS='A' you can have two servers that can run jobs in the same application, but one selects messages with JOBCLASS='A' and the other server selects messages where JOBCLASS is NOT A and all the A-class job go to the first server.

One caution – message property names can't contain dots, so things like com.ibm.my.property won't work as property names (thus the underscores in the application name property).

The Multi-Server Batch Configuration – Work Throttling

Assume we've configured a Liberty server as a batch dispatcher. We also have a batch message queue defined in MQ. Finally, we've got two servers configured as executors, both defined to select messages for our one batch application. We submit a job to the dispatcher, which causes a message to be placed on the queue. Which server runs the job?

You can't tell which one will get it. It just depends on which one MQ delivers the message to. Ok, fair enough. Suppose you submit two jobs at basically the same time. Will one job go to each server or will one of the servers get both? Again, it depends how MQ handles message delivery to the activation specifications. Well. Fine. Suppose I submit thousands of jobs in a burst. What happens then? Ah…that we know.

Nothing good. MQ will start delivering the messages to the servers the servers will start spinning up threads to run the jobs. If the jobs aren't really short, more and more messages will arrive while the first jobs are still running. More threads will get created. Eventually one (or both) servers will just be overwhelmed with threads and running jobs and bad things will happen.

Can we prevent this? Sure! Part of the configuration of an activation specification is the ability to set a limit on the number of messages that are being processed by the server at one time. This brings up an interesting point – while the job is running, the message is "being processed". The server isn't done with the message until the job finishes. Which allows us to solve our overrun server problem.

By setting a limit on the number of messages being processed at once, we can limit the number of jobs (for that activation specification) being run in the server at once. Careful selection of this value will allow the server run however many jobs it is able to run, but not more than that.

The configuration used to set this limit is slightly different if you are using the integrated messaging provider that comes with Liberty or if you are using IBM MQ. You can see IBM Techdocs paper WP102600 for examples of both.

My Job is Urgent – Batch Job Message Priority

We've talked about the multi-server configuration for Liberty that allows jobs to be placed on a message queue and processed by Liberty servers acting as job executors. And so far, we've just assumed that those messages are processed in the order they are put on the queue. Well, they are, but that's only because we haven't done anything special to change it.

By default all the messages representing jobs are put on the queue with the same priority. That means they get processed in the order they are received. But JMS supports a message priority. If you could specify a different priority for the message representing the job, you could influence the order in which the jobs (messages) are processed.

By default the message priority is set to four. The allowed range is zero to nine, where zero is the lowest priority and nine is the highest (that will be hard to remember because it makes sense..). When a JMS application puts a message on a queue, it can specify a message priority. But in this case, the batch container code in Liberty is the JMS application, so how do you get it to specify the priority you want? By using a special, magical, job parameter. Remember that when you use `batchManager` or `batchManagerZos` (or the REST interface directly) to submit a job you can specify job parameters as name/value pairs. The magic parameter name for job priority is `com_ibm_ws_batch_message_priority`.

If you set that job parameter to a numeric value (like 9) then the message, and thus your job, will be assigned that priority. A job (or message) with a priority of 9 will jump ahead of all the other jobs on the queue with lower priority. Of course jobs with the same priority are processed in the order they are received so a priority of 9 may put your job in line behind all the other priority 9 jobs..

By default a restarted job uses the same job parameters as the original job, unless you override them. That means a restarted job will have the same priority as the original job, unless you deliberately change it.

But wait, there's more! What if you want your job to wait to run until later? You want to submit it now, but delay it actually running until midnight or some other time. Turns out there's JMS support for that too. When a message is put on a queue you can specify a delay (conveniently in milliseconds) until the message should be delivered. And Liberty conveniently surfaces that ability as another special, magical, job parameter called `com_ibm_ws_batch_message_deliveryDelay`. Just get out your calculator and figure out how many milliseconds are between now and midnight (more or less) and there you go.

Unlike message priority, a delay value is not carried forward to a restart of the job. That would be confusing.

One last caution.. if you are using IBM MQ as the messaging engine an extra system queue has to be defined to give MQ somewhere to put the message while it is delayed. See the WebSphere Knowledge Center for details.

So if your job is urgent (or not) you can make it run sooner (or later).

The Multi-Server Batch Configuration – No Server? No Problem!

Once again, suppose we have a dispatcher putting jobs on a message queue being handled by two executor servers. Suppose both servers have message selectors defined to process messages for an application called MyApp. And finally, suppose one executor's selector adds that it only wants MyApp application names where a job parameter of JOBCLASS=A was specified, and the other executor's selector only allows JOBCLASS=B.

Got that? So, if we submit a MyApp job with JOBCLASS=A it goes to the first server and if we submit a MyApp job with JOBCLASS=B it goes to the second server. Fabulous.

Now we submit a MyApp job and specify JOBCLASS=C. What happens? If it is a typo, it would be nice if the job just failed. But it might not be a typo. It might be that you do indeed have a server that processes JOBCLASS=C jobs for the MyApp application. But they are only active from midnight to four in the morning (and it isn't in that window right now). In that case you'd like the job (the message representing it really) to just hang out in the queue until midnight when automation starts the server for JOBCLASS=C and then run the job.

And that's what happens. If no server can be found that has a message selector string that will match the properties on a message, that message will just wait in the queue until something turns up.

In cases where the mismatch is intentional, that's awesome. It allows you to submit jobs to run in servers that aren't presently active, and the jobs will just wait around in the queue until the server gets started to handle them.

Or you could do something really cool. When the job is submitted and put on the queue batch event messages are published into the batch topic tree. A monitor could subscribe to those topics and, being aware of what servers are up and what properties they handle, could recognize a job that didn't match the current set of servers and (on demand) start up the appropriate server needed to handle the job.

Such a monitor would also be handy in case the mismatch really is a typo. If you submit a job with JOBCLASS=1 (where all job classes are letters) that job is just going to sit in the queue forever. A clever enough monitor might recognize that as a problem and generate an alert.

The Multi-Server Batch Configuration – Partitions

We're still talking about dispatchers and executors, but let's go back for a moment to our discussions about the Java Batch specification and recall how partitioned steps work.

A partitioned step allows you to run multiple copies of the same step concurrently. You can specify different values for the properties that get injected into the elements of the step (such as the `ItemReader`) for each partition. This allows you to tell partition one to start at record zero and process to record 1000, partition two to start at record 1001 and process to record 2000, etc.

Assuming contention problems or system resource limitations don't cause problems, you can significantly reduce the elapsed time required to process a set of data by processing different ranges of it concurrently on separate threads.

But what if one server doesn't have the capacity (heap, or CPU available on the system where the server runs) to actually run the partitions concurrently in an effective way (or maybe at all!). Wouldn't it be cool if we could get the partitions to spread out across several servers?

Ok, now remember all that dispatcher/executor stuff we've been talking about. It turns out that if you add some configuration to an executor that allows it to also act as a dispatcher, we can do this. When an executor server reaches a partitioned step, it checks to see if it can also act as a dispatcher for partitions. If it can, then a message is created for each partition and put on the specified queue.

Same queue as the job queue or a different one? Turns out it doesn't matter. Either way works, so do whatever makes sense for you. There's a property added to the message by dispatchers and executors-acting-as-dispatchers to indicate whether the message represents a 'job' or a 'partition'. This allows you to configure executor servers to just select jobs for a given application or just partitions or both.

You might have three tiers of servers: dispatchers, executors that run jobs, executors that run partitions spun off from those jobs. And remember that executors and dispatchers communicate through the message queues and the Job Repository, but don't have to be co-located on the same OS image. Which would let you spread partition messages to servers scattered across different physical systems and separate heaps, letting you run more, larger, partitions concurrently, hopefully reducing overall elapsed time.

The Multi-Server Batch Configuration – Bad, Bad Messages

One more time – suppose we have a Liberty server configured as a batch dispatcher and another configured as an executor. We use the REST interface to submit a job to the dispatcher who puts a message on the queue which the executor picks up. Then….a bad thing happens.

Ok, it depends on the kind of bad thing. Generally speaking if the message gets picked up and the job starts to run but something goes wrong then the job will be marked failed and the message is considered processed.

What's that about the message being processed? Ok, remember that for throttling to work the message is 'being processed' by the server the entire time the job is running. Most things that can go wrong will result in the Job Repository being updated with the job marked as failed and the batch code will finish processing the message and all is well.

But. There are some scenarios where bad things can happen along the way to executing the job, before the Job Repository is updated to know the job started dispatch, and an exception gets thrown clear out of the batch code and the message is unhandled. What happens next depends how the messaging engine handles these cases and possibly how it is configured.

Some messaging engines will handle a failure to process a message by just re-delivering it. That's really awesome if whatever went wrong was some transient thing and the jobs gets to run on the second try. But what if the problem is in the message itself? What if there's just something, somehow, wrong with it?

It may be possible to configure a backout threshold for the batch job queue in the messaging engine (such as MQ). If it is possible, this tells the messaging engine how many times to attempt to re-deliver a message that failed being processed before giving up on it. You can also specify a backout requeue queue where such bad messages get parked.

This configuration of the messaging engine will prevent an endless loop of the server trying and failing to process the same bad message over and over. Instead it will try a few times (backout threshold value) and then give up. The message is handy in the backout requeue queue so you or your support organization can look at it to figure out what the fuss was all about.

Security Overview

Security is a large and complex topic. Securing the batch environment properly is going to be important if you plan to run your Java Batch applications in a production environment. The next few posts will try to hit a few of the more interesting points that you might want to consider. These posts are, of course, by no means the definitive word on proper security set up.

So what sorts of things do we need to worry about? We'll assume you've got the server(s) themselves set up properly and able to correctly access their own configuration and runtime libraries. We do still want to worry about access to the server to submit and manage jobs. This is an operational consideration that can get pretty complicated.

We'll also want to make sure the batch container code can access the resources it needs (the Job Repository and possibly a job message queue). And we need to make sure nobody else can get to those resources to muck around in them.

And finally, we'll want to be sure we've handled the batch application's access to resources it needs. This might include files or databases or other resource managers. Control of access to those will depend on how the resource is accessed and it might matter what identity the job itself is running under.

If your servers are publishing batch events into the batch topic tree, access to subscribe (and publish) will be important.

And remember that joblogs are being written into the server's file system, so access to those directories will need to be controlled.

In subsequent posts we'll take a closer look at a few of these topics. There are, of course, many more. You might have a look at IBM Techdoc WP102696 for more details and examples.

Securing the Job Repository and Job Queue

The Job Repository is where the batch runtime keeps track of everything that is going on. It knows about jobs that have run in the past as well as jobs currently executing. It has application checkpoint data to be used if a failed job needs to restart.

In order to function properly, the batch runtime that is part of the Liberty server needs to be able to update the tables that make up the Job Repository. As with much of the security stuff we'll discuss here, exactly how you do this depends on the platform you are running on and the database implementation.

Of course, it isn't enough to just allow access to the tables by the server itself. You want to deny access to those tables to anyone that shouldn't have access. Since all operational batch activities should go through the REST interface and the batch code, there's no real reason for anyone (except perhaps a DBA) to have any access to the tables directly. No one should need to go poking around in the tables to find the state of a job, and certainly no one should be manually making updates to the table content.

Similarly, in a multi-server batch environment where jobs are being queued from a dispatcher to an executor server, access to the queue itself will be important. Just as with the Job Repository tables, the server itself needs to be able to write to (dispatcher) or read from (executor) the queue to put/get messages (technically, since it is a "destructive" read, the executor needs write/update access too). And also, like the Job Repository, you don't want or need any others being able to directly access the queue, with the possible exception of an MQ administrator. And, yet again, exactly how you do this will depend on the messaging implementation and the platform.

Finally, if you configure the servers to publish event messages to the batch topic tree at significant points in job processing, you'll need to allow the servers (both dispatchers and executors) to publish into that topic tree. You'll also need to identify any necessary subscribers that are interested in batch events and allow them to subscribe. It is important to block out any access from unauthorized users as you certainly don't want the possibility of 'fake' events introduced into the system by an unauthorized publisher. And, since the event messages can include job log contents which could contain sensitive information (depends what the batch application logs) you'll want to control subscriber access also.

Batch Roles

The ability to submit and manage batch jobs through the REST interface has several layers of security. The first is simply the ability to make an HTTPS connection to the server and execute under an authenticated identity.

But after that, what controls access to jobs? Well, what kinds of things do we need to do? We want to control who can submit a job, but we also want to control who can look at the results of those jobs and manipulate then (e.g. stop and purge).

This access is controlled by defining roles in the security configuration for this server. This can be done directly in the Liberty server configuration itself or externally in another security repository.

The batch code defines three main roles (plus two more we'll get to later). The first role is that of a submitter. As a user with permission to this role you can, naturally enough, submit jobs. But it also allows you to look at and control jobs you submit. That's important. A user who submits a job but can't find out what happened to it will be a frustrated user. This makes the role of submitter a little special because it has full operational control over its own jobs. If all your production jobs are submitted by one functional id, that ID has a lot of power over the jobs.

The second main role is that of monitor. A monitor essentially has read access to the job information kept in the repository. A user acting as a monitor can find out the status of jobs, retrieve job logs, and generally look at all the job information. But a monitor can't change anything. They can't stop a running job or submit or restart jobs. They just watch.

The final main role is an administrator. A batch administrator can do all of it. He can submit jobs. He can look at job status and job output. He can stop running jobs and purge old jobs. Be careful who you grant access to the batch administrator role.

Next time we'll look at those two other roles I hinted at….

Group Based Security

Last time we talked about the submitter, batch monitor, and batch administrator roles that can be used to control access to operational control of and information about Liberty batch jobs. It might have struck you that monitor access is nice, but read access to every job in the system might be a bit much. And an administrator really just has way too much power. It would be nice if there was a way to scope that power to a subset of the jobs.

To do that, we'll first need to find a way to identify subsets of jobs and indicate who should have control of them. Suppose we have a whole bunch of executor servers hosting many different applications. Some servers host one application, some host several related applications. Since they are grouped by application (more or less) that's probably how we'll create groups of access. There are probably groups of developers or operations people that need access (read or write) to things related to certain applications but not to others.

To specify this in the configuration, as part of the `jmsBatchExecutor` configuration you can specify a list of security group names. These are the groups that have a reason to need access to jobs handled by this executor. Remember that the `jmsBatchExecutor` element points to an activation specification which can include a message selector string that can limit which application jobs execute in this server. So the control of which-applications is tied to the specification of which-groups need access.

But what access? Some groups might need just read access, such as a developer needing to look at a log. But other groups might need more, such as an operations staff needing to purge or restart a failed job.

To support this, two additional roles are defined: `batchGroupAdmin` and `batchGroupMonitor`. If a user is allowed to be in the `batchGroupMonitor` role AND is a member of a group associated with job (from the executor configuration) then that user can have batch monitor access to the job. But users who are not connected to that group (i.e. in a different development department) won't be able to look at jobs associated with different departmental groups.

Similarly access to the `batchGroupAdmin` role allows administrator access, but only to jobs associated with groups that user is a member of.

Careful use of these two additional roles can allow more access to those who need it, without opening the doors to everything.

Batch Application Security

When a batch job runs in a Liberty server there is an identity associated with the thread running the application. That identity is the same identity as the user who submitted the job. If the REST interface was used to submit the job, that means the identity that authenticated over the HTTPS connection will be the identity used to execute the job.

That identity is probably some functional id used by an enterprise scheduler to submit every job. That identity probably does not have access to all the resources needed by the job. How can we get around this?

Well, it depends on the type of resource the job is trying to access. Database access done through a datasource definition in the server configuration can often include access credentials in the configuration to be used instead of the identity running the application. This works just the same for batch applications as it does for any other application in Liberty.

In some cases, the application identity is used, but the submitter's identity is the wrong one. In these cases, it might be possible to define a RunAs identity for the application. This is a Java EE security concept that allows, as part of the application packaging and server configuration, the specification of an identity to be used to run the application that isn't the identity that submitted the job. You can do this sort of thing with non-batch applications deployed in Liberty. The same thing works for batch applications.

Other resources might use the server identity, ignoring the identity running the application. In some cases, these are operating system controlled resources (i.e. a z/OS dataset). The identity of the server itself will be used to grant or deny access.

If that's the wrong identity, it might be possible to configure the server and application to perform what is called 'sync to OS thread'. This causes the server to take the identity running the application and 'push' it down onto the native operating system thread that is underneath the Java thread. In some cases, the OS or other system resources will honor this native-thread identity in preference to the identity of the server itself.

There's clearly a lot more to this. Hopefully I've given you some hints to follow in solving whatever your particular problem might be.

Extra Things Just for z/OS

Applications that could probably be classified as ‘batch’ can likely run on every platform in existence. Java applications can run on a huge assortment of platforms. Java Batch applications can (and do) run on a lot of platforms. But…. z/OS (OS/390, MVS, MVT, OS/360..) has been running batch applications longer than anything still actively used in real production environments. And z/OS offers a lot of very special capabilities that Enterprise Batch environments like to take advantage of.

We’re going to start a series of posts that look at some special things done in the WebSphere Liberty implementation of JSR-352 just for z/OS. Some of these are direct exploitation of z/OS capabilities, such as SMF. Exploiting SMF provides the ability to produce reports detailing information about batch job execution, such as CPU consumed. This information allows for chargeback for use of system resources, but also for capacity planning as batch jobs grow to consume more resources (if you have a batch job that processes a record for every widget you sell and your number of sold widgets doubles, then your batch job takes longer to run and consumes more resources – it would be nice to see that growth happening).

Other z/OS exploitation features take advantage of z/OS capabilities to simply fit in better with a z/OS environment or take advantage of z/OS features to execute faster or more simply.

We’ll also take a look at how do to some things on z/OS that are slightly different than you might do them from a batch job on another platform. This will include things like accessing datasets (which are basically files but can be…different). We’ll also look at some special considerations for managing file locking (e.g. GRS ENQ contention).

Batch applications written in Java can run on a lot of platforms. You might have reasons why you want to run those applications on z/OS and we’ll talk about how you can do that. And we might even give you some reasons to run on z/OS if you’re not sure that’s a good idea (hint: I think it is a good idea…)

Stay tuned!

The z/OS Native Command Line Interface

A while back we talked about a command line interface (CLI for short) called `batch-Manager`. This is just a little Java program that uses the Liberty Batch REST interface to submit jobs and perform other batch-related operations. On z/OS there is a second CLI called, conveniently, `batchManagerZos`. Why have two?

There's nothing wrong with the Java CLI (`batchManager`). But given a z/OS environment there are a few things that could be made easier or simpler. First of all, since `batchManager` is written in Java, you have to start up a JVM to run it. In a z/OS environment a Java Batch job is likely being run from inside some JCL. One step of that JCL job will be used to submit the Java Batch job (more on this topic next time). Well, if you're going to have to start a JVM in order to submit the job, why not just start the JVM and run the batch work right there? That makes the first difference between `batch-Manager` and `batchManagerZos` the fact that the z/OS specific CLI is written in 'C' and thus doesn't require a JVM.

It is also often the case that the JCL job submitting the Java Batch job is running on the same z/OS system as the Java Batch job will run on (or at least where a Liberty Batch Dispatcher is located). That means going through a TCP/IP connection to drive a REST interface is a bit heavy. WebSphere Liberty supports the WOLA (WebSphere Optimized Local Adapters) interface which allows direct cross-memory communication between the client and server. The `batchManagerZos` CLI uses WOLA to communicate with the target Liberty server.

Using WOLA means the communication doesn't flow over TCP/IP which means it doesn't need to be encrypted which allows you to avoid the configuration required for SSL. That simplifies the setup. The WOLA protocol also just 'picks up' the identity of the client and propagates it to the server (lifting it from the associated ACEE). That means the client doesn't have to supply a userid/password as part of establishing the connection.

Of course, SOME security set up is required to enable WOLA and allow the client to access the server using WOLA services.

And, since the 'L' in WOLA stands for 'Local', `batchManagerZos` can only communicate with servers on the same z/OS system. It can't reach across the sysplex to a server on a different z/OS system.

Next time we'll look at integrating `batchManagerZos` calls into a JCL job.

Integrating with JCL Driven Jobs

We've talked about how Enterprise Schedulers can run scripts that run `batchManager` to submit Java Batch jobs to WebSphere Liberty. But what if you want to run JCL on z/OS? How do you integrate the two together?

Both `batchManager` and `batchManagerZos` are command line interfaces that can be invoked from JCL using the BPXBATCH program (`EXEC PGM=BPXBATCH`). Or you can use one of the other flavors of BPXBATCH such as BPXBATSL (which might be a better choice).

However, you probably don't want to invoke the CLI directly from JCL. The reason has to do with return code handling. BPXBATCH itself has return codes to indicate errors with its own processing (such as not being able to find the thing you told it to run). Those return codes values are likely to collide with return values from the thing it ran (your program, or in this case `batchManagerZos`). To avoid that, BPXBATCH shifts the executed program's return codes over one byte. Thus, a return code of 0x8 (hex 8) becomes 0x800. That's all well and good if the return codes from your program are small values.

If your application (or `batchManagerZos`) returns 0x10 then that gets shifted to 0x1000 which is all fine, except that JCL condition codes are only 3 digits (hex) which means the actual JCL condition code for a step that used BPXBATCH to run a program that returned 0x10 is going to be…. Zero. That high-order '1' gets lopped off.

Well guess what – the return codes from `batchManagerZos` are often bigger than 0xF and run into this behavior.

The answer is to use BPXBATCH to run your own shell script which, in turn, runs `batchManagerZos`. The shell script can examine the return codes from the CLI and convert them into its own return values that are less than 0xF and can be handled in JCL condition code processing as values from 0xF00 down to 0x100 (with values less than that being return values from BPXBATCH itself – or zero…zero is always zero no matter how much you shift it : -)

For a sample to start with, have a look at the `WASdev/sample.batch.misc` project in GitHub. You may need to adjust that to meet your own conventions or handle exit status values from the batch application itself (for which you hopefully have some standard established).

Using the JZOS Library and Accessing Datasets

Today's topic isn't really specific to writing JSR-352 Java Batch applications in Liberty. It is just a question that often comes up when writing Java applications intended to run on z/OS. In fact, as I'm writing this, a question popped up about it on IBM-Main.

So, you have some pile of data that exists in a z/OS dataset. Maybe it is a sequential dataset or a member of a PDS. Or maybe you've got a VSAM dataset that's been in use forever. How do you access that stuff from Java? Regular Java file access methods are going to look for files in the USS file system, but you could copy your dataset there. And Java file access methods are going to assume ASCII encoding (which probably isn't the case for you – although you can address this if you're careful). And, of course, it doesn't help at all with VSAM.

JZOS to the rescue! Originally created by Dovetail Technologies but long-since included in the z/OS JVM, is the JZOS library. JZOS is actually two things. There's a JVM launcher which provides a handy way to spin up a JVM and run Java from JCL (simpler than using BPXBATCH to run a script that starts a JVM). But JZOS is also a set of Java libraries usable from any Java program running on z/OS.

To access datasets (VSAM or otherwise) you'll want to have a look at the ZFile class which provides methods to do all sorts of z/OS file operations. One thing I've learned to be careful about is the allocation of new files using ZFile. There's no provision (at least that I know of) to specify much in the way of file attributes, especially size. If you need to allocate a new, really big, file then you likely won't get what you want just using the defaults ZFile will provide. To get around this, ZFile includes an API called bpxwdyn which wrappers the USS BPXWDYN API which is really just a wrapper around native z/OS dynamic allocation (SVC 99). Using that successfully to allocate a larger dataset (or one with other special attributes outside the defaults ZFile provides) will require some specialized knowledge and probably some research.

JZOS also provides a long list of other z/OS specific APIs to let you do things like issue WTOs (if your batch applications interact with your automation) and obtain ENQs (which are sometimes used to coordinate activity between clever, concurrently running batch applications).

Take a look…there's some cool stuff in there.

Dataset Contention Issues

Traditional JCL-based z/OS batch applications will go through an allocation phase that establishes access to datasets (files) used by that application. This generally results in shared or exclusive access to the dataset. Access might be established using JCL (DISP=OLD, SHR, or NEW for example) or programmatically using dynamic allocation services.

Batch jobs need to establish the right level of access to prevent other concurrently running programs from accessing the data (or to allow it). Two jobs that both require exclusive access to a file can't be allowed to run at the same time.

Key to all of this is the fact that the lock (really a GRS enqueue) will be released at the end of the job (possibly sooner). This is done by the operating system at the end of the job.

Java batch applications running inside WebSphere Liberty introduce some complications. A dataset allocated by an application running inside a Liberty server is generally done using dynamic allocation (not using DD statements in the server's JCL). A dynamically allocated dataset will be automatically released when the address space terminates – which is not the end of the job, it is the shutdown of the server.

Thus, any datasets allocated by Liberty Batch applications should be sure to close them to release the allocation. Be sure to do this, not just in normal termination cases, but in error/failure cases where you may need to rely on a step listener or other artifact to be sure your application gets control to perform the close processing.

Another complication arises when a Java Batch job in Liberty is submitted by a JCL batch job (as we've discussed earlier), but both the Java Batch job and the JCL job have steps that require access to the same dataset. Perhaps a step in the JCL job populates a dataset and a step in the Java Batch job reads the contents. If the JCL job uses a DD statement to allocate the dataset, it will remain allocated throughout the JCL job (usually) which includes the time the Java Batch job is running. If both require shared access, this is just fine, but if one or the other requires exclusive access the two will deadlock when the Java Batch job can't get the dataset access it requires, and the JCL job is waiting for the Java Batch job to end.

There are several strategies to work around these situations, but it gets rather complicated to go into as a blog post. Instead I'll point you to the WP102667 whitepaper on the IBM Techdocs website which delves into this whole topic in detail.

What About Utility Programs – DFSORT & IDCAMS

Multi-step batch applications consist of a lot of steps that run your application code. But sometimes there are utilities that you have to use. How do you invoke those from inside a Java Batch application? You might be tempted to blend JCL jobs that run the utilities with steps that submit Java Batch jobs, which will work. But beware of the dataset allocation issues we talked about last time.

Before you try that, you might check for a Java interface to the utility you need to use. Conveniently, the JZOS library provides interfaces to both DFSORT and IDCAMS.

DFSORT is the IBM sort product, although under the covers it might be replaced by another product – the JZOS API doesn't care. You can use the sort utility to do all sorts of very clever things, often very fast.

IDCAMS is the utility for managing VSAM datasets and ICF catalogs. A quick search online tells me that IDC is the IBM product prefix and AMS stands for Access Method Services. DFSORT apparently stands for Data Facility Sort…I always thought of it as my personal sort (starting with my initials…).

Anyway…. How does this help you? Well, having a Java API that drives the existing utility means that you could create a batchlet that calls the utility. Where you need a step in a JCL job that calls the utility, you simply have a batchlet step that calls the API that calls the utility.

But what about parameters? Both utilities support a pretty extensive (and sometimes lengthy) input syntax to tell it what to do. Normally that is just instream data in the JCL. How do you provide that to a batchlet that is calling an API?

Remember that any batch artifact can have string properties set in JSL that are injected into the artifact (the batchlet in this case) at runtime. If we set a property in the JSL that is the input string to the utility, it will get injected into the batchlet and be able to be passed to the utility. You might want to do this as a JSON string to make it easier to work with.

If you'd like to see more detail, check out WP102636 in IBM Techdocs which provides samples. Of course, for production use you'd want to 'productize' these samples to handle errors and such in a better fashion. But it is a start.

SMF Recording

If you're running a batch workload on z/OS, you're worried about (at least) two things: how long (elapsed time) do the jobs take and how much CPU the jobs use. This might be because you're trying to keep the batch workload inside a particular part of the day (in the infamous "batch window") or for financial reasons (four hour rolling averages, internal chargeback, or whatever). Tracking everything you need to know about your batch workload is most easily done by processing the SMF Type 30 records z/OS writes for the batch initiator address space where the job ran.

But what about Java Batch jobs running inside a Liberty server? You will, of course, still get Type 30 records for the server address space. That doesn't help very much though because one server could run multiple jobs…concurrently.

WebSphere Type 120 records to the rescue! Liberty writes SMF 120 subtype 12 records for Java Batch jobs. A record gets written at the end of the job, at the end of each step, at the end of each partition, and the end of each flow. Like the type 30's you'd examine for traditional batch applications, the 120-12 records contain information about elapsed and CPU time. They also contain information about how the job or step ended (batch and exit status values) plus information to help you identify the specific job (and step, flow, etc).

What they don't contain that the type 30's have is information about I/O and memory usage. For a traditional batch job, the job occupies the entire address space, so all I/O and memory usage belongs to that job. Liberty Java Batch jobs are sharing a Java Heap and the rest of the resources of the address space and it isn't possible to attribute usage of those shared resources to a particular job.

CPU accounting for Java Batch jobs isn't even simple. Remember our multi-threading capabilities with partitions and split/flow constructs? The CPU recorded for a job is actually only the CPU used by the 'main' job thread. To find the total CPU for the whole job you really need to hunt down the end of partition and end of step records and sum up the CPU time from those to determine the total CPU time used.

Why doesn't Liberty do that for you? Well, remember that partitions can spread across multiple servers which could be located on different z/OS images which could be running on different physical hardware. To calculate the correct 'total' CPU for the job you need to normalize the different CPU times. Do you trust us to do that right? Didn't think so….

SMF Accounting Data

Last time we talked about the SMF 120 subtype 12 records that are written for Java Batch jobs run inside WebSphere Liberty. A topic we didn't get to was how you know who submitted which job (which might be related to who gets the bill for the CPU used by the job!).

A common technique is to just map the job name to the owner. The job name for a Java Batch job is the 'id' tag in the job element of the JSL. Remember that this is just a string and can be longer than the eight-character length of a traditional JCL job name. You might also want to recognize the job based on the actual name of the XML file packaged with the application that contains the JSL. Both of these approaches assume some sort of internal standard for naming these things (and that people follow the standard…).

You might also want to tie it back to the JCL job that submitted the Java Batch job. If your Enterprise Scheduler submits JCL jobs that use `batchManagerZos` to submit the Java Batch job, then the jobname and id of the JCL job will automatically be propagated into Liberty and included in the Identification section of the SMF 120-12 record. That might fit better into the ownership mapping scheme you already use.

The identity of the submitter of the Java Batch job (which might be the identity running the JCL job in the above scenario) also turns up in the SMF 120-12. That might help, unless you have a functional id that runs all your batch workload.

Another option is the accounting string. Traditional JCL allows an accounting string to be provided in the JOB card and the values will turn up in the SMF Type 30 records. For Liberty Java Batch applications you can specify a job parameter whose name is `com.ibm.ws.batch.accountingString` and the value will be used to set the contents of the Accounting section in the SMF 120-12 record.

The value of that string is parsed apart at commas and each resulting sub-string is used to create an accounting section in the SMF record. For example, if your accounting information consists of a department number and name, you could specify an `accountingString` whose value was "12345,CEOs Office" and the SMF record will have two accounting sections, one with the string "12345"and the other with "CEOs Office".

If you have a standard script that uses `batchManagerZos` to submit jobs, you can build into it the specification of accounting string value(s) as part of your standard processing.

Exploiting a Split to Stop a Chunk

The intent of this post is to just explore an idea I had that might help solve a very specific problem with certain batch jobs. Suppose you have a job with a chunk step. For whatever reason, your `ItemReader` can sometimes get ‘stuck’. It reaches out to wherever it gets data from and doesn’t come back. You would like to be able to stop the job when this happens.

The `stop` operation does two things: it marks the job as stopping in the Job Repository, it drives the `stop` method in a batchlet if it is what is currently running in the job. For a chunk step nothing gets control. The assumption is that the read/process execution loop will come around to a check on the job status, notice the job is stopping, and stop without continuing the loop.

But in our scenario, that isn’t going to happen because we are stuck in the `ItemReader` trying to read an item. But suppose we have a handle or something to the connection we are using to read data. And suppose there is some sort of ‘cancel’ operation we can call on that connection to try to terminate an outstanding request. That would be nice, but the chunk won’t get control to issue it. There is no `stop` method in a chunk like there is in a batchlet.

Suppose that we wrap our chunk step in a flow. And suppose that flow is part of a split with a second flow. And the second flow consists of a batchlet. The batchlet doesn’t do anything. It is just there for its `stop` method to be driven in the event a stop is issued. That `stop` method could then cancel the hung read operation (if there is one).

For this to work, you would need to share whatever the connection handle thing is that is used by the reader in one flow with the batchlet in the other flow. You would also probably want some flags and locking so the batchlet’s stop method can tell if it is supposed to try to cancel an operation or not.

You also need a way for the chunk step to communicate to the batchlet that it has finished and the batchlet can exit.

Is this a recommended pattern or some other official thing? Nope. Just me pondering how you might shake loose a stuck chunk step. Other thoughts or ideas?

Pipelining with a Split-Flow

One of the things that some batch applications try to do is reduce execution time is by “pipe”-ing the output of one step into another one and running them concurrently.

The idea is that one step is processing records and producing results and there is a second step that needs those results to do its own processing. Why wait until the first step has finished and processed all its input before starting up the second step to feed on the results from the first step (so far)?

Well, the batch spec doesn’t specifically support this because steps run in an order. But with a split/flow you can get different steps running concurrently without needing separate jobs. Yes, you could do what I’m describing with separate jobs too, this just seemed cooler.

The idea is pretty simple. The job has a split. The first flow in the split is a chunk step that reads data from some source and does some processing. At checkpoints the resulting records are put in a queue (whether a real message-queue or just some file-based implementation that acts like one). The second flow in the split has a reader that reads records from the queue, does its own processing, and writes its results to a second queue. A third flow reads from the queue being filled by the second flow, etc.

At each checkpoint in each flow the results from that chunk are pushed to the next flow which can start processing them. It isn’t quite a smooth pipeline as records move between the flows in chunk-size piles. But it beats waiting for each step to complete before moving along.

You will need to carefully consider how this ends. There needs to be some sort of all-done message that goes on the queue to tell the secondary flows to complete. And be sure to get that message passed along in case an error causes one flow to shut down early. Also consider how a restart of that failed job will handle finding messages in the queue from a previous run if the flow that handles them dies early.

The general flow of this could be pretty generic. Might make a nice sample. Check out IBM Techdoc WP102784!

The Retryable-Skippable Exception

I was recently reminded of a rather specific paragraph in section 8.2.1.4.3 of the JSR-352 specification. I've quoted it here:

When the same exception is specified as both retryable and skippable, retryable takes precedence over skippable during regular processing of the chunk. While the chunk is retrying, skippable takes precedence over retryable since the exception is already being retried.

What does that mean? Let's consider an example. Suppose your job is processing records with nice integer record numbers. We're working with an item-count of 10 to keep things simple. The job reads and processes records 1-10 without a problem. The writer is called with the processing results, and checkpoint processing completes.

Then we start the second chunk with record 11. All goes well until we begin processing record number 13 (because it is unlucky). An exception is thrown by the processor which is listed in the JSL as both a skippable and retryable (with rollback) exception. What happens next?

The current transaction is rolled back (undoing anything done by the reader or processor handling things since the commit of the first chunk). The reader and writer are closed and re-opened with the checkpoint data from the previous chunk. We begin again at record 11. Since we are in retry processing the item count changes temporarily to one. That causes the step to read/process/write for record 11 and commit the one-record chunk. This happens again for record 12.

Unlucky record 13 again has a problem and throws the same exception. This time we're in retry processing so the special rule above applies. The exception is handled as a skippable exception and we skip the record. The chunk will complete and retry processing ends (because this is the record that caused the retry-rollback processing to start).

After that we're back to regular 10-item chunk processing starting with record 14.

So what? Well, specifying an exception as both skippable and retryable allows you to rollback and redo processing for records up to the problem one, but then continue on skipping the problem record if a retry at handling it fails again.

Without the skippable specification for the exception retry processing would kick in again and we'd make another try at record 13. If it continues to fail, we'll just keep retrying until we hit the retry limit for the step and the step fails. That assumes you specified one…you did remember to specify a retry limit when you set up retry processing, didn't you? The default is no-limit which is probably not what you wanted.

Monitoring a Chunk Step with a ChunkListener

Once in a while I get involved with a batch application that has a chunk step that is not performing as well as the developer had expected (or perhaps has management had hoped). Sometimes you can fiddle around with the chunk interval and improve things. Partitioning may or may not be an option. Ultimately everybody just ends up wondering what the heck is going in there…. How can you tell?

Well, the obvious thing is to put in some message logging and see. But developers are often reluctant to clutter up the application with monitoring code. Fortunately, the batch programming model supports listeners that let you do that without touching the application.

It occurred to me that, with the various listeners you can configure for a chunk step, it would be possible to implement all of them (or most of them anyway) in a single Java class and accumulate metrics about what was going on in the step. You could record elapsed time (and, depending the platform, CPU time) for each read, process, and write.

While you could log start and end timestamps as you go, that's going to clutter up the job log something fierce, and probably slow things down a bit. Remember that while one logged message might not cost much, one logged message every time through a loop that runs several million times will cost a lot.

Instead, just hang onto the information in an in-memory table and dump it all out to a file when the step ends (because there's a step-end listener you can implement too!). Enabling or disabling the listener(s) is just a matter of putting the elements in the JSL – no application updates are required beyond having the listener class in the deployed application .war file. There's no contact between the application and the listeners, so they don't need to be in the same Java package. And you can have multiple implementations of any listener get control in the same job, so there's no worry about these listeners causing problems for listeners used by the application.

Then just run the job as usual and have a look at the resulting data. Are you spending your time reading data? Processing data? Writing results? Is it all the time or are there "special" records that seem to take longer? Or maybe it is time-of-day related…perhaps every 10 minutes access to a database slows down mysteriously…something else is touching it and causing contention? Implementing this monitoring listener won't tell you why it is slower than you expected, but it is quite likely to at least give you some clues to pursue.

Oh, and I went ahead and wrote up the listener. You can find it in github here: https://github.com/WASdev/sample.batch.misc and you can find a whitepaper that goes into more detail about how it works here: http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102780

Handling Changes to Checkpoint Data

As a chunk step reaches a checkpoint, the reader and writer are both called to provide checkpoint information. This has to be a Java Serializable object. The serialized data is hardened into the Job Repository as part of the transaction commit processing at the end of the chunk. In a retry-rollback or job restart scenario, that checkpoint data is retrieved and provided to the reader and writer as they are called to perform open processing.

What if a job fails and before it gets restarted the application is updated and that update includes a change to the contents of the checkpoint data? What happens? Does the job fail? How do you handle this? Or can it just never change?

This is really just a case of a common problem with hardening Serializable Java objects. If it is possible for the application to change between serialization and deserialization the application needs to handle it. There are numerous posts elsewhere that discuss the general topic, but I'll try to just make a few suggestions.

Firstly, you should add a serialVersionUID to your checkpoint data class. Change it only when you make incompatible changes to the class contents. An exception will be thrown trying to deserialize the object if you try to read a version of the class that is different than the one currently loaded.

It is possible to make compatible changes to a class if you're careful. For example, you can add a new attribute to a class if you remember that deserializing an older version will set that field to its default value. Be sure to check that a newly added String isn't null before you try to use it if it is possible an older serialized version might still be around.

You can also implement your own readObject and writeObject methods to make sure things get properly initialized if the default values cause you problems. You can also bury your own class version number as an object attribute if that can help you figure out how to handle the serialized object. Of course, that only works going forward (a version 2 class can read a version 1 object, but not the other way around). That could happen if you ran a job that failed and then rolled the application itself back to a prior version before restarting it.

There's a lot of different advice online about techniques to handle different types of changes. Of course, the best approach is to just never change the data or else be sure you don't have any failed jobs that need restarting before you roll out an application update that changes the checkpoint data.

Creating an Aggregate Reader

The specification allows a chunk step to have a Reader, a Processor, and a Writer. But what if your application needs to read from more than one source? Suppose you have a table containing client information (name, address, email, etc) and another table containing loyalty-award-point information. Both tables use a customer id as the primary key. You want to write a chunk step that iterates over all the clients pulling out the customer information and award point information so that processing can create customized email text to be sent out by the writer.

Of course, you could just do that by having the reader get the client information record and the processor fetches the matching award point information, but lets say we want to do it all in the reader.

You might already have `ItemReader` implementations that can read from both tables. What you really want to do is put two readers in the JSL for this step, but the spec doesn't allow that. What to do?

The idea is to create a sort of generic aggregate reader that calls the other two readers and aggregates the results into a combined object to be passed to the Processor. That sounds simple but turns out to be rather complicated.

First of all, the aggregate reader will get all of the properties that would normally be injected into both readers. You'll need to sort out some syntax for the properties so the aggregate can tell which ones go to which reader. Injection won't work for the reader's called by the aggregate, so you need some interface into the existing readers to pass along the values. A Properties object would work well.

The next thing to consider is checkpoint information. Each reader will have its own and the aggregate will need to merge them together into a single serializable object and be able to separate it back out in a restart to pass the right information to each reader's open processing.

The results of the two items read will need to be merged into a joint object to be passed to the process, but that's just the usual arrangement to work out between a reader and processor.

Finally(?) there are error cases to consider. What if one reader fails for some reason? What if, for some reason, one table runs out of rows before the other one? Maybe a customer has no award points.

As you can see, this is an interesting idea but tricky to implement. It would be especially difficult to do as some sort of generic aggregation reader with no specific knowledge of the existing reader implementations it is calling.

Initialization Processing for Batch Artifacts

The ability to inject properties from the JSL into batch artifacts makes it possible to avoid hard-coding values and can let you dynamically change how a job behaves without changing the code. A problem I run into occasionally is trying to inject a value that I want to use as part of initialization for the artifact.

The problem is that property injection doesn't happen until after the constructor for the artifact runs. For the reader and writer that's ok. Open processing is as good a place to do initialization as the constructor. You just have to be careful not to do it again if you are handling any kind of retry/rollback processing where open would get called again.

But what about other artifacts like a batchlet, or the processor, or any of the listeners? For a batchlet it doesn't really matter. Just do everything at once when it gets called. But the processor and most of the listeners can get called a lot. Let's look at the processor.

Besides the constructor there is just one method that gets control over and over. You could, of course, just have a Boolean called initialized that starts as false and lets you do your initialization the first time through the `processItem` method. That works except that you're going to check that Boolean every time through the read/process loop and only do initialization once. That's millions of times (potentially) that you're going to do that pointless check and those little bits of CPU usage add up.

Another approach might be to do the initialization in the `beforeStep` method of a Step listener. You could hang the results in a Thread Local and access it from the processor with confidence. Unless your step is partitioned in which case each partition will have a separate Thread and thus a separate Thread Local.

There's no batch artifact that gets control just once at the start on each partition thread. With a partitioned step you're pretty much stuck checking some Boolean every time through the loop.

You might be tempted to use class statics to anchor things, but remember that more than one copy of a job from the application could be running concurrently in the server at the same time. And it wouldn't help anyway if your environment is configured to run partitions in separate servers (using the Liberty partition dispatcher/executor model).

Now that we have things initialized, what about cleanup? That's for next time.

Cleaning Up After a Step

Last time we talked about doing initialization processing for a step. This week we'll clean up after ourselves. We'll take the easy cases first.

As with initialization, cleanup for a batchlet is simple. Initialize when you get control and clean up before you return. For a chunk step (or even a batchlet) you can also use the Step Listener to initialize and clean up in the before and after Step methods.

As with initialization, where this gets tricky is in a partitioned chunk step. There is no point of control for each partition that gets control after the partition is complete that runs on the partition thread. There are before and after methods in the chunk listener, but it gets control after every chunk and there's no way to know which one is the final chunk. Or is there?

How does a chunk step know that it is done? The partition map provides parameters to each partition that are usually used to let each partition know what it is supposed to do. The common example is the start and end of a range of data this partition is to process (record numbers or primary-key values, etc). The reader for each partition will have those partition values injected and will use them to find a starting point and read records until reaching whatever end value it was assigned.

Which might make you think about having something in a chunk listener that also got those values injected and somehow knew which record the reader was on so it would know where we are in the processing for this partition. But it is actually easier than that.

When the reader reaches the end of its assigned range of things for this partition to do, it has to indicate to the batch container that it is done. That's handled by having the reader return a null instead of an object it read. The batch container sees that null return value, skips the call to the processor, and completes the final chunk for this partition. Could we somehow signal between the reader knowing it has run out of data and the after chunk listener?

Well, of course we could have some shared object that could do that. But a simpler approach is to have a single class that implements the Read Listener and the Chunk Listener. The Read Listener will get control after each read completes and can examine the returned object. If it is null, the partition is done. The listener can set an object attribute state flag. When the after chunk method in the same class gets control, it can examine that flag and know that this is the last time through the loop and do whatever cleanup is required.

Batch Performance – Introduction

This post will kick off a short series talking about performance topics related to Java Batch jobs using JSR-352 (also known as Jakarta Batch, but I'm going to stay out of that for now). What do we mean by batch performance? There are standard bench-marks that are used to measure performance of various online transaction processing systems. Is there something similar for batch? Not that I could find. And that wasn't where I wanted to go with this. Instead I wanted to talk about decisions you might make as you are developing a Java Batch application that could impact how it performs.

We're just going to look at a single step job to keep things simple. The performance of a batchlet step is pretty much up to how you write your application and things that would apply to any Java program will apply there. On the other hand, a chunk step has some interesting design decisions you have to make that influence how the batch container behaves and probably influence the performance of the step.

For example, how often should your chunk step take a checkpoint? It seems pretty intu-itive that if you checkpoint frequently you are introducing more overhead and the job will run longer. But how much longer? How frequently is too frequently?

In the coming weeks we'll take a single step job running a chunk step and fiddle around with how it behaves to see how those changes affect the elapsed time to run the step. But we have to start somewhere!

As a baseline we'll use an ItemReader which reads records from a flat file. We won't do any processing (except for one measurement) because processing time is just whatever your application needs to do and there's no special batch stuff about it. Our writer will alternate between inserting records into a DB2 database and deleting those same rec-ords. That saved me from having to remember to clean up the database between runs. However, I did the inserts as a bulk insert and the deletes were done one-by-one which allowed me a rough comparison of the two techniques.

I also included the chunk step listener we talked about a couple of weeks ago (post #101) to get more details about what's going on inside the chunk processing.

Our baseline numbers had the reader fetching 10 million records from our flat file. The initial item count (chunk size) was 1000 records. The checkpoint data size for both the reader and writer was 1024 bytes. Batch events were not enabled. To start I ran it as a simple step without partitions (but we'll get there).

All the runs were done on a z/OS system (z14) using a local DB2 accessed over a type-4 connection. The same DB2 instance was used for the Job Repository and the appli-cation table.

With all that, our step took four minutes and not-quite 33 seconds to complete. Next time we'll start playing with checkpoint intervals.

Batch Performance – Checkpoint Intervals

From last week, we have a chunk step that reads 10 million records from a flat file and then alternately inserts and deletes records from a DB2 database. Using an item count of 1000, the step took four minutes and 33 seconds to run on our performance test system.

What we want to know is what happens if we increase or decrease that item count value. For an item count value of 1000, our step takes 10,000 checkpoints. We dropped the item count down to 500, then 250, then 125, then 75, and finally 25. For that final test the step took 400,000 checkpoints. You would think that would make the step take a lot longer to run…and it does.

Lowering the item count to 500 (twice as many checkpoints) added about 30 seconds to the elapsed time to run the step. Dropping it to 250 added another full minute. By the time we got down to 25, the step took almost 24 minutes to complete.

Clearly increasing the number of checkpoints taken can greatly increase the elapsed time for the job. But what happens if we go the other way? If we increase the item count to 2000, will the step run faster? Turns out that it does, but not by as much as cutting the item count in half did. In this case the step ran about 16 seconds faster.

We kept increasing the item count size up to 7000 and saw progressively small improvements. At 7000 the step took just over 1400 checkpoints and ran in about four minutes and four seconds.

At this point I should emphasize that this data is only relevant for the application I was running and in my particular environment. Your results may (and probably will) vary quite a bit.

That said, it does seem safe to conclude that cutting the item count value way down dramatically increases the number of checkpoints that are taken and those do come at a cost. We can also see that there is some limit to the benefit of increasing the item count to reduce the number of checkpoints. A happy value for my situation seemed to be around 5000.

Another factor to consider when choosing a checkpoint interval is how hard it will be to recover from failure. If my chunk step were to fail on item 4999 of a 5000-item-count chunk any updates would have to be rolled back. DB2 is going to do that automatically for me, but if your application and environment require some manual intervention to roll back changes…well..that's a lot to roll back by hand. You might want to consider accepting a bit longer elapsed time for the step to reduce the pain when something bad happens (the key word there being 'when' instead of 'if').

Next time we'll look a bit closer at where our chunk step is spending that time.

Batch Performance – Database Updates In Bulk or 1-by-1?

This week we'll take a look at the performance of our itemWriter. Remember that it alternates inserting rows in one chunk and then deleting those same rows in the next chunk. This was mostly to avoid having to remember to clean out the database between runs, but it also gave me a chance to explore the performance difference of doing a bulk insert vs doing one-by-one deletes. Of course we're comparing apples and oranges here because insert and delete processing is different. I was more interested in how the two things scaled.

Let's take a look at the conventional approach where we did the deletes one by one. We're processing 10 million records and deleting half the time, so in total the step deletes 5 million records from DB2. The commit size varies depending on our item-count value. At an item-count of 25, the total time spent doing deletes was about 185 seconds. With an item-count of 1000, we spent 209 seconds doing deletes. And scaled all the way up to 7000 records per commit, we spent a total of 213 seconds doing deletes.

Increasing the commit size caused the step to run a little bit longer. The improvement from 25 to 1000 was 24 seconds which isn't a lot, but it helps. But pushing it farther didn't really buy us much (just 4 seconds). In the end I think it doesn't matter a lot how many deletes you do in each chunk. Deleting 5 million records is going to take some time.

What about the bulk inserts? That's a different story entirely. In the alternating chunks we're going to insert a total of 5 million records, but all in one JDBC interaction that inserts however many records are in our item-count size. Starting again at 25 I found that the step spent about 42 seconds doing inserts. Scaling up to 1000 items at a time the inserts cost 20 seconds. That's a huge improvement. Ramping up again to 7000 items the elapsed time only dropped to 18 seconds. Clearly there's a limit to how much bulk inserts buy you.

Looking at the data for the entire step it was clear that by far most of the elapsed time for the job was spent in the writer. Remember the whole step took four minutes and 33 seconds. It depends on the item-count value, but in general our writer spent around 200 seconds (3 minutes 20 seconds) doing deletes. While inserting rows only took around 20 seconds once we got away from low item-count values.

The breakdown at an item-count of 25 is interesting. Remember that run took about 24 minutes. As noted above, we spent 185 seconds doing deletes and 42 seconds doing inserts. The rest of the time was spent in 'container overhead' (around 1207 seconds). The container overhead dropped to just 41 seconds at an item-count of 1000.

Bulk interactions with a database are better. Container overhead decreases dramatically as you get past small values. In general (for my application!) an item-count value of around 5000 seems pretty good.

Batch Performance – Checkpoint data size

For the next experiment I wanted to explore how changing the size of the serializable checkpoint data returned by the reader or writer impacts the elapsed time of the job. At every checkpoint the batch container gets this information from the reader and writer and updates the relevant row in the Job Repository table as part of the chunk commit scope. A minor change in elapsed time, multiplied by a large number of checkpoints, could theoretically impact the runtime for the job.

In theory the state data required to restart from a checkpoint shouldn't have to be very big. But if you're using a non-transactional resource you might have to do some rollback-type cleanup yourself and that might require more data.

And, of course, sometimes objects just accumulate information. I've seen HTTP session state objects that were several megabytes.

The question we're looking at here is how much it matters. Of course, there are a lot of other factors at play. Checkpoint data is hardened in the Job Repository which is just a database. The main consideration is how quickly that database row update can take place. In my experiments I was on z/OS accessing a local DB2 instance over a type-4 connection. Switching to a remote database is going to require checkpoint data to flow over a physical network which is going to significantly slow things down.

Note that while the default size for the BLOB that contains the checkpoint data is 2GB, your DBA might have pruned that down (mine did) and you'll get SQL errors trying to save very large checkpoint data (I did). Remember that both the reader and writer checkpoint data are placed together into one column. It is easy enough to alter the table to make it as big as you need, assuming you can convince your DBA you need the space.

For our runs we began with our same baseline, processing 10 million records in 1000 record chunks. I updated the checkpoint data size for both the reader and writer to the same value. Our baseline run uses a 1024 byte object so the total checkpoint data was 2048 bytes. We scaled that up and down by factors of 10. For each run, as before, we determined the reader and writer time from the chunk listener and subtracted that from the elapsed time for the step to determine the time spent in the container (which includes the time spent updating the checkpoint information in the Job Repository).

The results:

Data Size (bytes)	Container Time (ms)
20	41851
2048	42271
20480	45636
204800	66454
2048000	303026

Which suggests that you can have pretty large checkpoint data objects without significantly impacting the elapsed time for a chunk step. Only at really large sizes did it start to matter (2MB and up). In the end, don’t put stuff in the checkpoint data that doesn’t need to be there, but don’t obsess about it.

Batch Performance – Process result size (heap utilization)

For this experiment we wanted to see how changing the size of the object returned by the ItemProcessor affected how the step ran. Our baseline application doesn't have an ItemProcessor at all, but we'll add one in for this set of measurements.

The ItemProcessor I created established a large byte array when the constructor ran and then made a copy and returned it every time the processor was called. Naturally making a copy of a very large byte array consumed quite a bit of elapsed time, but we wondered how it would impact elapsed time for other processing in the chunk because of the increased heap usage.

Remember that the reader and processor are called in a loop until we reach a check-point. Our baseline run checkpoints every 1000 items. The writer is only called at the checkpoint. Which means the objects returned by the processor (all 1000 of them) pile up in memory until they are passed to the writer in a single list.

To get started we used a returned object size of just 1000 bytes. This added about 5 seconds to the total elapsed time for the step from our baseline run with no processor at all. Then we increased the object size to 100,000 bytes. The time spent in the proces-sor to copy a 100,000 byte object 10 million times was pretty significant, but not interest-ing.

What was interesting is that time spent in the ItemReader increased by six seconds. Time spent in the ItemWriter (which isn't actually processing all those objects..it is still just inserting and deleting rows) increased by 20 seconds. Time spent in the batch con-tainer increased by 16 seconds. That's all due to garbage collection running to handle the 1000 objects returned by the processor each checkpoint.

We took the verboseGC output and fed it to the GCMV tool and you could very clearly see that there was minimal GC activity with the 1000 byte object, but a frenzy of GC ac-tivity with the larger returned object.

This really shouldn't come as a surprise to anybody. Increased thrashing of the heap drives garbage collection which interferes with the efficiency of application code.

You should pay attention to the size of the object returned by the ItemProcessor to make sure it doesn't get too crazy. Think about your checkpoint size and how many of these objects are going to accumulate in memory. Also consider that you might have more than one job running at the same time in a single JVM. Factor all that into deter-mining your heap size and then use tools like GCMV to look at how things are actually running and tune accordingly. It matters.

Batch Performance – Partitions

The next item of interest was how use of partitions can reduce elapsed time. In this experiment we used the same reader/writer (no processor) we started with and continued to process 10,000,000 records with an item-count value of 1000.

What we changed was the level of concurrency. In all our runs so far we used a single thread to process all the records. This time we'll split up processing of the records across multiple threads. We'll begin with just two partitions each processing 5,000,000 records and proceed to 10 partitions processing 1,000,000 records each.

Our partition mapper arranged for each partition to use different primary key value ranges for the database inserts/deletes to avoid contention. In a real application we'd hopefully be inserting (or deleting) different rows, but how the table is locked (row etc) could impact you.

Remember that just because we have 10 partitions doesn't mean all 10 will run concurrently on 10 separate threads. Instead 10 work items will be queued up inside the server and will be picked up when a thread is available to run the work. Liberty will dynamically adjust the number of threads it has to try to handle the work, but we don't pre-configure the server with some number of threads.

After 10 partitions, we'll proceed to 20 and then 100 and finally 1,000 partitions each processing a single chunk of 1,000 records.

From the results we could see that the first increment to two partitions nearly cut the elapsed time in half (137 seconds down from 272). Moving to 10 threads cut the time to not quite 1/10 of the single threaded version (32 seconds instead of 27). But 20 partitions almost cut the time in half from 10 partitions (down to 17 seconds). But our gains began to max out as going to 100 partitions didn't even cut the time from 20 partitions in half (about 10 seconds). Why is that?

Most likely it is because the server can't spin up 100 threads very fast and instead just re-used existing threads to process different partitions sequentially. And when we moved to 1000 partitions the step actually took longer than with just 20 (23 seconds).

I should point out that these experiments were run on a dedicated LPAR. Nothing else was running on this OS image, nothing else was touching DB2. The system I was running on was a z14 with 20 GPs. Clearly a real-world test would involve some contention for resources and affect the results.

Nevertheless, it seems quite clear that, when possible, the use of partitions can dramatically reduce elapsed time for a step.

Batch Performance – Batch events

The JSR-352 implementation in WebSphere Liberty includes the ability to publish messages into a topic tree at interesting points in the lifecycle of a job. Messages (or events) are published around the start and end of a job, as well as the start and end of each step. Most interesting to us is the event published at each checkpoint. In our baseline job there will be 10,000 checkpoints and thus 10,000 messages published. Does enabling this feature (which allows a monitor to track the status of the job) significantly slow down execution of the job itself? Let's find out.

We can configure the server to publish batch event messages by pointing it to a messaging engine. We can use the WAS messaging server built into Liberty or point to an instance of MQ running on our z/OS system. If we use MQ we can connect over TCP/IP or cross-memory (Client or Bindings mode).

For my experiment I choose to run using the built in Liberty messaging services located in the same server running the batch job. I thought that would probably interfere the most with the running job and be slower (going through TCP/IP) than a Bindings mode connection to MQ.

Remember that we're comparing to our baseline run with no ItemProcessor, handling 10 million records, checkpointing every 1000 items. That step ran in about 4 minutes and 32 seconds. For our test I ran the exact same configuration except that the server was publishing messages as the job progressed. This time the job took 4 minutes and 42 seconds.

So there is clearly a cost, but not a very big one. There is no way to configure the server to only publish messages for selected events (e.g. skipping the checkpoint messages). A very long running job that had hundreds of millions of checkpoints (instead of the 10,000 we had) might see the job run a few minutes longer with events enabled.

Jakarta Batch Issues

If you've been reading these blog posts for a while you probably know that JSR-352 is the Java Batch specification that's part of Java Enterprise Edition (Java EE for short). You might also know that it is now called Jakarta EE. There's a whole lot involved in that which I'm not going to get into. But the bit we care about here is that JSR-352 is now Jakarta Batch. The official home is here:

The home for the source of the actual specification is now in github here. And being in github means we can have issues (which is 'github' for any defect, suggestion, or other thing you'd like to have changed). To just lay the groundwork for any future enhancements to the spec, we opened up issues for all the things we had, either in the old repository or just tucked away in notes and other things. And doing all that gave me a chance to read over some of them and I thought maybe it might make for some interesting blog posts.

Over the coming weeks I'll be selecting some issues from the list at the link above and just talking about what it means, why we might want it, and maybe how it might work. I should be clear, first of all, that talking about it here in no way indicates a preference for that particular item or that it has any more or less chance of getting into a future release (or even suggests that there might be one of those). If you're interested in any of that sort of thing, I'd suggest you get involved in the online project. This is all open source, so everybody can help.

Well, if none of those are the reasons why I am talking about particular issues, how did I pick them? The first criteria was that it was something I thought might make an interesting blog post. That immediately eliminated all the 'typo on page 86' issues…

The second criteria was that it was something I understood. I'll be honest and say that there are a few of the issues that I don't really understand. Either I don't follow the problem trying to be resolved or I don't understand how the proposed solution would address it. There are a few that just state a problem and don't have any proposed solution. I left those out too.

Again, that doesn't mean there's anything wrong with those issues or that they have any better or worse chance of being included in a follow-on release to the current specification. I just thought it would make a more interesting blog post if I understood what I was talking about (you can stop laughing just any time now).

Just one more time…if you find some proposal in the following weeks that makes you say "Wow…I wish we had that" get involved in the project and help make it happen.

Jakarta Batch Issue 133 – Batch Properties Object

This issue can be found here.

The current behavior allows you to specify a batch property in the JSL and have that value injected into a batch artifact like a Batchlet. Batch properties are all strings. When you inject a property you give it a name and a value. To get the value injected into the Java code you need to know the name given to it in the JSL.

For example, in the JSL you might have this:

```
<properties>
     <property name=”outputType” value=”PDF” />
</properties>
```

And in the Java code you would have this:

```
@Inject
@BatchProperty(name = "outputType")
String inputDir;
```

That makes the JSL and the Java code pretty tightly wound together. It would be nice if the Java code could just pick up a Properties object that had whatever properties were specified in the JSL and then extract the keys to find out what properties were specified and react accordingly.

It would also let you group a set of related batch properties together. That would help if, for example, your reader was reading from two different data sources. You could have two groups of properties, one for each source, that would be injected into two Properties objects, keeping them together without requiring some naming convention or something.

The JSL might look like this:

```
<properties>
     <properties-object name=”Prop1”>
          <property name=”outputType” value=”PDF”/>
     </properties-object>
     <properties-object name=”Prop2”>
          <property name=”outputType” value=”PDF”/>
     </properties-object>
</properties>
```

And then in the Java code you could have something like this:

```
@Inject
@BatchProperty(name=”Prop1”)
Properties prop1;
@Inject
@BatchProperty(name=”Prop2”)
Properties prop2;
```

Of course that’s just a proposal and it might have to change to actually work.

Jakarta Batch Issue 57 – Partition Listener

The issue can be found here.

This time we'll take a look at a requirement for another listener. This one would have been helpful a few weeks back when we were talking about initialization and cleanup for application artifacts used by a step. Before we get into it, let's have a quick review of listeners.

The batch specification defines quite a few different listeners that can be implemented to get control at different points in the execution of a job or (mostly) a step. There's a job listener that gets control at the beginning and end of the job. But for each step you can also define a listener to get control too. A chunk step has a bunch more listeners around the reader, writer, and processor, plus other listeners to handle error conditions for skip and retry processing.

Interestingly, for a step you just specify the class that implements a listener as part of the step and the batch container will determine which listener interface it implements and make sure it gets control at the right times. You don't have to specify in the JSL that this listener class is a Read Listener. It implements that class, so that's what it is.

You can, of course, have a single class implement multiple interfaces and the listener method names are all carefully chosen so they don't interfere. You can have one class that implements all of the listener interfaces if you want to and the batch container will make sure the right methods given driven appropriately.

The specification is also very clear that there is no particular order to drive when you have multiple classes that implement the same listener interface for the same step (or for the job).

So what about the partition listener? Well, as it stands right now in the specification there is nothing that gets control around each partition like a step listener does for the whole step. There's a chunk listener that gets control for each chunk (if the partitioned step is a chunk step). It would be nice to have something that got control on each partition thread before and after each entire partition.

The only way to emulate that behavior now is with a chunk listener with special logic to help it figure out if this is the start of the first chunk or the end of the last one and emulate before/after partition behavior. Just having that as part of the specification would avoid having that logic in the application.

This one should be easy enough to implement by adding another interface and putting the appropriate processing around execution of the partition. At least it sounds easy to me…I could be wrong…

Jakarta Batch Issue 165 – Whole Job Step Listener

The issue can be found here.

One more on the topic of listeners. We'd mentioned before that there is a job listener that gets control at the beginning and end of the entire job. We've also got the ability to configure a step listener for each step.

The scope of the step listener is just the step that is defined for. Remember that all the listeners for the step are just listed as listeners and the batch container figures out what type of listener it is from the interfaces it implements. You could define a dozen listeners for the step and they might all implement the step listener interface, among others. So any listener defined to the step is a listener of some sort for that step.

Back at the job level you can define listeners, but the only implementation you can have there is the job listener.

The proposal here is to allow a step listener to be defined inside a listener element at the job level. That listener would get control before and after every step in the job.

It is an interesting idea because right now each step is mostly considered to be an independent thing. The steps are joined together in a job, but the job listener is the only artifact whose scope spans the entire job.

What would you use a step listener for that could run before and after EVERY step? I suppose it could look in the Step Context to determine what step it was in and behave differently, but then you've really just got a different listener for every step that happens to be in the same Java code. All you've saved is some JSL specifying it everywhere. That could be valuable by itself, but seems a bit thin as justification.

Would you really have common processing that would apply to every (or at least most) steps? Maybe some logging processing? An update to job status somewhere when every step begins and ends (although you could also trigger that sort of thing from the Liberty batch-events support)?

I suppose it is possible there could be some common application processing that has to happen between every step in the job (or at least most of them). Maybe after each step you want to make a backup copy of the data being worked on or something like that.

Feel free to add some use cases to the issue if you can think of something.

Jakarta Batch Issue 143 – Get Partition Number

The issue can be found here.

If you want to run a partitioned step (running multiple copies of the step at the same time) then you have to create a partition plan. You can do that either statically in the JSL itself or use a Partition Mapper to create one dynamically. The plan consists of properties that will be provided to each partition. Typically, these properties help each copy (partition) know what range of data it is supposed to operate on.

When you define the plan you specify partition numbers (starting with zero) for each partition and assign (probably) different property values based on the partition number. Thus, the partition number is a known thing to the batch container based on information you provided in the partition plan.

And so you might reasonably expect that there would be some way to determine from inside the running partition which partition number you are. You might want to create some sort of output file from within the partition and you want to make sure each partition uses a unique file name. An obvious place to get a unique value would be to use the partition number. If only you knew what it was.

Turns out there is no way to get the partition number. The batch container knows and could certainly provide it, probably as a method on the Step Context. Would it throw an exception if you called it from a non-partition thread?

In the meantime you can get around this by simply adding a property to the partition plan that is set for each partition to the partition number. You'll feel a little clunky doing it because it seems (at least in retrospect now) that this was sort of an obvious thing to need.

This strikes me as one of those reasonably valuable and yet pretty easy to implement enhancements that should really be added to the specification.

Join in and let us know how you feel about this issue or any of the other ones starting from the link above. We're always looking for help in defining and implementing enhancements to the batch specification.

Jakarta Batch Issue 140 – Job Parameters in Job Context

The issue can be found here.

When a job is submitted the submitter can choose to specify job parameters. These parameters are name/value pairs where both are Strings. It is essentially a Properties object.

Typically these parameters are used to set job properties. Job properties are defined in the JSL in a properties element and, again, consist of name/value pairs where the value is always a String. The syntax allows for a default value to be provided if the job parameter on which the property is based was not set by the submitter.

Job parameters are frequently used as values for properties of other batch artifacts and injected into them at runtime. For example, a chunk step might have a reader that needs to know the file name to read from. That file might be provided to the reader as a property whose value comes from a job parameter (either directly or via a job property set from the job parameter).

A batch artifact can also just access the job properties directly using the Job Context. The getProperties method will return a Properties object containing all the job properties specified in the JSL (and possibly some extra ones depending on the implementation).

However, there's no way to get to the actual job parameters that were specified by the submitter. You have to have JSL that assigns them all to job properties. Which means that the JSL has to be aware of all the possible things the submitter might specify.

The proposal here is to add a method to the Job Context that would return a Properties object with the entire set of job parameters specified by the submitter. This could eliminate a lot of extraneous JSL that is just there to convert parameters into properties.

We would probably want to allow implementations to tuck in extra parameters so you shouldn't be surprised to find things in there that the submitter didn't explicitly specify.

We'd also want to be clear (as we are with getProperties) that the object returned isn't the actual job parameters object, so you can't add/delete/change things in it and expect other batch artifacts in the same job to see your updates.

Sound interesting? Follow the link above and join in the discussion.

Jakarta Batch Issue 94 – Flush API for Persistent User Data

The issue can be found here.

As usual, let's review. The Step Context has methods on it that allow an application to set and get what is called Persistent User Data. The data itself is just a serializable object. The spec says that the data is persisted at checkpoints. For a step that doesn't do checkpoints (a batchlet) it is persisted at the end of the step. The data is available on a restart.

What does that mean? Well, basically it provides a notepad where the application can put things that it might want to remember (besides the reader and writer checkpoint data) for a restart. This could be something from the processor or maybe something from one of the listeners that needs to be available for a restart.

Well, if the data is persisted at checkpoints along with checkpoint data, why would you want a flush method to force it to persist outside that scope? I suppose it is possible you might have data you need to remember that isn't tied to the chunk processing. That seems rather specialized.

The real use case is for a batchlet. We normally think of a batchlet just doing some piece of processing that isn't in a loop. If you need to do it in a loop, you'd do it with a chunk, right? Well, not always. There might be reasons you want to manage the loop yourself. Or perhaps this is code that used to be run as a stand alone Java program (just driven from a main()) and it was easier to convert to a batchlet than rewrite into a chunk.

Regardless of how you got there, you might find yourself mid-batchlet and need to persist something in case the job fails. That way when the job gets restarted your batchlet can get the persistent user data from last time and pick up where it left off or do whatever it needs to do with the data it remembered.

Presumably this would just be another method on the Step Context.

It does feel like there might be some complexities about this as it interacts with the automatic persistence occurring in a chunk, but I can't see what it would be. If you can think of something feel free to comment in the issue linked above.

Jakarta Batch Issue 129 – Set Persistent Data by Reference?

The issue can be found here.

When a call is made to set the persistent user data, is a copy of the user data made or just the reference kept? That's the question raised by this issue. What's the difference?

Let's suppose we have some serializable class we defined to hold our user data. We're in the Item Reader and we set some fields in an instance of this object. The reader calls the API to set the persistent user data.

Suppose this instance of our user data object is accessible to our Item Processor. If the processor changes something inside the object data, what happens? If the user data object reference was held by the batch container, then updating the data inside the object updates the persistent user data. When we reach the end of the chunk and commit the chunk transaction, whatever happens to be in the user data object at the time will be persisted.

But what if the batch container makes a copy of the user data? Then the update made by the processor won't have any effect on the copy of the data kept by the container and only the initial values set by the reader will be persisted.

Of course you can get consistent behavior by just calling the Set API every time you make a change to the user data. But that's rather clunky, especially if it isn't necessary. In fact, it might be handy to just have to set the user data object once at the start of the step and let the batch container just persist whatever happens to be in there at the end of each chunk. Then the batch application can just update it as needed without having to worry about remembering to Set it.

The problem raised by this issue is that the specification doesn't actually say which behavior is required. This leaves implementations free to do either thing. Which essentially requires you to assume a copy is being made and call Set every time you change the data just to be safe. If the implementation you develop with decided to keep a reference and you depend on that, your application might not behave correctly when run on a different implementation. And that was certainly NOT the goal of having a specification!

The point here is that a decision should be made as to which behavior should be in the specification and TCK tests should be written to verify it.

Have an opinion? Follow the link above and comment on the issue!

Jakarta Batch Issue 131 – Persistent User Data and Rollback

The issue can be found here.

The question here is what happens to the in-memory copy of the persistent user data when a chunk step performs a rollback. Remember first how a chunk step works. Once things are going a transaction is started and the reader and processor are called in a loop until a checkpoint is reached. Then the writer is called to write the results of the processing, checkpoint data is collected, and the transaction commits.

If the application sets persistent user data then that data is pushed into the Job Repository along with the reader/writer checkpoint data and that update becomes part of the transaction commit processing.

What happens if the step encounters an error that results in the transaction rolling back instead of committing? The current in memory persistent user data isn't pushed into the repository and the checkpoint data is rolled back. The reader and writer get the previously committed checkpoint data provided to them when their open methods are called after the rollback.

But what about the persistent user data? Suppose we are keeping a count of records processed that match some criteria. Every time through the processor we might update the value and (to be safe) call the Set method to set the persistent user data. At every checkpoint the commit will harden that count to go with our checkpoint data (that is keeping track of where we are processing records). That's all good.

If an error occurs and the transaction rolls back, the reader will pick up again at the record just after the last checkpoint. But the specification says nothing at all about what happens to the persistent user data. The copy in the Job Repository didn't get updated, but what about the current in-memory value? We may have called the Set API a few times since the last checkpoint to change the data, but does the contents of the current user data according to the batch container reflect what it looked like at the last checkpoint (rolling it back) or is it just whatever the last set value was which was from part-way through a chunk that rolled back?

It seems pretty clear that it would be a lot of work for an application to manage this itself and so the batch container should probably make the current persistent user data be whatever was set before the last checkpoint committed. But the specification never actually says that. And it probably should.

What do you think? Follow the link above and weigh in..

Jakarta Batch Issue 147 – Commit User Data after Mapper

The issue can be found here.

A few weeks ago we talked about the possibility of adding an API to force a commit of the persistent user data. That would allow a batchlet to harden the user data whenever it liked. In the absence of that API, there was some discussion about other places in the flow of a step where you might want to have the data persisted.

Remember that in a chunk step the user data is persisted along with the transaction that wraps chunk processing. From the wording it would appear this does not include the transaction wrapped around open and close processing, but only after every chunk. Would it be useful to have it updated there? There isn't an open issue for that, but it is interesting question.

The idea raised by this issue is to have the persistent user data automatically committed into the Job Repository after the partition mapper runs for a partitioned step. Remember that a partitioned step can be made up of either a batchlet or a chunk step with multiple copies running in parallel.

The partitioning can be statically defined in the JSL, but you can also do it dynamically by implementing a Partition Mapper. The mapper returns a set of properties to be passed to each copy of the step. For example, each partition of a chunk step often gets a property telling it which record to start reading from and which record to stop at.

On a restart of a job, the partition mapper can choose to use the same properties from the previous execution or set up new ones. There might be some information you'd like to remember to help you decide what to do. For example, if you could remember how many items you needed to process and it was the same on a restart as the earlier execution then you could probably use the same partition map. But if the number of items has changed you would re-calculate. To make that decision you need some way to remember how many items there were across a restart.

Of course you could just put it in a temporary file or something else that you'd have to manage, but that's exactly the sort of thing the persistent user data is intended to help with.

Having an API to persist the data whenever you liked would be handy, but absent that API change it certainly feels like after the mapper runs would be a nice additional place to have the data automatically committed. Unless it causes some problem I'm not seeing.

Thoughts? Follow the link above and join the discussion…

Jakarta Batch Issue 95 – Set a Property from Earlier Results

The issue can be found here.

Every batch artifact can have values injected based on properties defined for that artifact. For example, a batchlet can have properties defined for it in the JSL and those property values can be injected into the batchlet at runtime. Where do the properties get their values?

The simplest way is to just set a value in the JSL itself. Or artifact properties can get their values from higher-level job properties. You can also get values passed into the execution by the submitter as job parameters. All these values are hard-coded except the job parameters and those are set before the job starts running. The exception is partition properties.

Each partition in a partitioned step can have properties assigned to it from a partition map created by the partition mapper that executes as part of the job. But what if you want to have the results of some prior step be injected into a later step in the job? There's no good way to do that with just JSL (you can, of course, side-step it by passing values directly between steps, but the batch spec isn't helping you do it).

The idea behind this issue is to have some way for a step to basically set a job property value on the fly. Then the JSL could specify an artifact with a property whose value comes from that job property. JSL parsing would have to accept that the property value wouldn't be known when the JSL is first parsed and so determine what value to inject later, like it does for partition properties.

A use case for this might be a step whose job is to find a file that needs to be processed by searching several directories. Once found it needs to provide that path/file to a subsequent step for processing. There are several ways you could do this, but it would be nice if the spec supported the first step setting that value somehow as a job property and the later step specifying its use as the value for its property.

How would a running batch application set a new job property? Well, we could just add a method to the Job Context. There is already a method there to fetch a Properties object containing all the job properties, so it would make sense to have an API there to let you set one.

There would have to be rules about what happens if you change the value of an existing job property. Would the old value already be substituted into JSL referencing it or would all job property substitution take place on the fly as needed step-by-step? Would the same late-substitution work for system properties whose values change during the execution of the job? Would just changing how system properties are handled be enough to address this whole issue?
Lots to discuss…

Jakarta Batch Issue 107 – Multiple Readers

The issue can be found here.

We talked about this idea a few weeks ago (post #103). The original specification allowed for one reader to be defined for a chunk step, but what if you need to read data from multiple sources to merge together (or something) in the processor? Sure you could build a single ItemReader that just reads from both data sources, but you might already have ItemReaders you regularly use that read from each source and it seems like you shouldn't have to write a third one to use them together. Maybe.

What would this look like? Would you just have two reader elements inside your chunk element? Presumably you could have more than two if we allow more than one. Is there an upper limit?

How would you know when the chunk step is complete? Would it end when 1-of-N readers returned a null or would all the readers need to run out of data? Maybe that should be an option for the chunk definition? Could it be a reasonable thing for a reader to just 'pass' on having an item this time around?

What about listeners? Would a read listener get control around each ItemReader or would we need some way to define which listener(s) go with which readers? Or at least a way to indicate to a listener which reader's results it was being passed? Or would a new read listener get control after all the readers were done and get their collected results in a list (like the ItemWriter)?

How does skip handling work? Does the skip limit apply to each reader or collectively to all of them?

What about metrics? Right now we count how many items were read. Would you need separate counts for each reader or sum the counts together?

The properties problem we talked about in an earlier post is solved here by each reader having its own set of properties.

This is clearly a pretty cool thing to be able to do, and it seems like a pretty common problem (multiple input data sources) to have. But as you can see there are a lot of details that need to be sorted out (and I'm sure I missed some).

If we allow multiple readers, should we also consider multiple processors? Writers?

As always, please feel free to join in the discussion at the link above.

Jakarta Batch Issue 126 – Add Methods to ItemProcessor

The issue can be found here.

The reader and writer have open and close methods as well as a method to gather checkpoint data before committing the chunk transaction. By contrast, the `ItemProcessor` just has one method, `processItem`. That method only gets control when the reader has returned data to process. This issue proposes some new methods for the interface that would let the processor get control a bit more often.

The first proposal is the addition of a `close` method to allow the processor to flush whatever state it might have. The original design of the processor didn't involve it having any state. It just gets control to handle the data provided by the reader. But in practice it is clearly possible for the processor to accumulate information such as a count of certain types of items it has processed. The processor might not exist to give any data to the writer, but just to count up items read that match some criteria. As defined there's no easy way for the processor to do anything with the final count because it won't get control when the reader is done reading (because the final read returns a null which means the processor won't get called).

The second proposal is to create an opportunity for the processor to get control at each checkpoint. It isn't clear from the text of the issue whether this would be an opportunity for the processor to provide checkpoint data or just a chance to get control at the end of each chunk. It seems to me that if we think the processor might be keeping a running count of things, it would want to checkpoint that information along with checkpoint information from the reader and writer. Of course a count is just a simple example, there are certainly more complex things a processor could do that would result in some state that would need to be persisted and aligned with the persisted checkpoint data from the reader and writer.

The issue doesn't mention adding an `open` method to the processor, but it seems clear to me that if we're going to add the other two, we should add this one also. It certainly seems reasonable that a processor might need to establish a connection (maybe to a rules engine or a database) to enable it to do whatever processing it is doing. There are other ways to do this, of course. We've talked about some of them in earlier posts. But adding a method to match the ones in the reader and writer would make it much easier.

Would the open and close methods for the processor be included in the transaction that is wrapped around those methods for the reader and writer? Probably, but there's an open issue about that also. We'll consider that one next week.

Meanwhile, feel free to join in the discussion on this issue at the link above.

Jakarta Batch Issue 160 – No Transaction Chunk

The issue can be found here.

This one is pretty simple. The standard chunk processing begins a new transaction at the start of each chunk and maintains it until a checkpoint is reached, either through the JSL specifying an item count or time limit, or through a checkpoint algorithm implementation determining that it is time.

When a checkpoint is reached the reader and writer are called to provide checkpoint information. The row in the Job Repository for this chunk (a whole step or maybe just a partition) is updated with the checkpoint data and the transaction is committed. Any transactional activity performed by the reader or writer (perhaps the writer inserted rows into a table) commits along with the updates to the Job Repository providing a consistent point across the application data and the checkpoint data.

But what if your application doesn't update any transactional resources? What if you just read from a flat file and produce some sort of report? What if you don't care about having a consistent restart point at all? Maybe you just re-run the job if something bad happens. Then all this transactional stuff is just overhead that slows things down and creates complications.

The proposal is simply that there be an option you can specify as part of a chunk step that indicates you don't want or need a transaction wrapped around the chunk processing. Just let it run….

Would the batch container bother to call the reader/writer to obtain checkpoint data in this case? Probably not. There'd be no reason to update the checkpoint data in the repository, although you could. Maybe that's optional on top of whether the main transaction is present or not.

What about metrics? There are metrics about the number of reads and writes and skips that happen that are updated at each chunk. Would those still get updated in a small immediately-committed update?

At some point you've removed enough value from a chunk step that perhaps it should just be a batchlet with the loop in the application code. At least that's what I think.

Chime in at the link above…

Jakarta Batch Issue 99 – Transactions Around Open/Close

The issue can be found here.

There is also issue 125 which is similar/related.

In the current specification for a chunk step a transaction is started and then the open methods for the reader and writer are called and the transaction commits. Similarly, a transaction exists around the close processing for the reader and writer together.

These items propose that these two transactions should be optional, or perhaps that there should be separate transactions for the reader and writer.

As we discussed last time, having transactions at all around open and close processing presumes some activity is going on in there that is transactional in nature and needs the commit to harden it. There is no activity by the batch container that needs to be coordinated with open or close processing that would need to be part of the same transaction. Which means the overhead of the transaction may be unnecessary and just clutter things up.

On the other hand, it could well be that there are transactional updates being made in open (or especially in close) processing. This seems most likely for the writer which may hold off making some final update until the close happens. Perhaps it needs to write a total count somewhere into a transactional resource.

For these situations a transaction makes sense, but why just one? Is it always the case that whatever the reader and writer are doing needs to commit together? Would some error in closing the reader preclude the writer from performing its close processing? An exception thrown from the reader's close processing shouldn't prevent the writer's close processing from happening and certainly shouldn't (necessarily) prevent whatever updates it might make from committing.

In this case we might like to have independent processing and transactions for the reader and writer open and close processing. But how would you indicate your preference? Presumably some syntax in the chunk element would do it. There are a lot of combinations that are possible: one transaction for reader and writer, separate transactions, a transaction for just the reader, or a transaction for just the writer. You might want to have all of those possibilities for both the open and close processing to allow for different behavior. Perhaps separate attributes for open and close with four values for each attribute. The default, of course, would be the original behavior.

Have some thoughts or ideas around this issue? Comment at the link(s) above!

Jakarta Batch Issue 152 – An Optional Writer

The issue can be found here.

When we talk about a chunk step we always talk about the reader and processor running in a loop and when we reach a checkpoint the writer gets control. But technically you don't have to have a processor. The JSL element doesn't have to be there. Well what happens?

If there is no processor defined then whatever result comes from the reader is accumulated as it were the output from the processor. When a checkpoint is reached, the list of those objects is passed to the writer.

This scenario basically covers the case where the reader itself is doing the "processing". Suppose the step is scanning some database table producing a report of all the clients who match some criteria. You could have the reader read the entire table and the processor filter out the results to be given to the writer. But you might be able to use a simple WHERE clause in the SQL used by the reader to restrict the results to just the ones you want. There's no processing to do. So the processor is an optional part of the chunk.

Which brings us to the issue at hand. The spec requires you to have a writer. But do you really need one? The only thing you really HAVE to have is the reader because it controls when the chunk loop ends (by returning a null). The writer could just do nothing at all but return. So technically it would seem that we could allow a chunk step to be defined without one.

But would you ever want to? Allowing this would permit the creation of a chunk step with just a reader and a processor, or even with just the reader and nothing else. Is there a use case for either of these?

Certainly the reader/processor combination makes sense. Suppose you're just hunting through a table looking for rows that match some elaborate criteria (beyond just a WHERE, maybe some other analysis is required). You don't actually need the specific rows, you just want to know if any exist. Perhaps there's some flow control in the job based on the results of this search. Then you could have a reader reading rows and a processor doing the analysis and setting the step exit status based on what it finds. There's nothing at all to write.

Would you ever have a chunk step with just a reader? I suppose you could. I start to wonder if maybe you would be just as well off with a batchlet, but perhaps there are cases where you'd want the transactional help when the step fails and you need to restart. Or perhaps there's something you could do with a reader, skippable exceptions and the skip listener.

Seems a worthwhile change to me. What do you think? The link's right up top…

Jakarta Batch Issue 163 – JSL Inheritance

The issue can be found here.

Your long lost uncle has passed away and left you a big pile of JSL in his will, but wait 'till you see the tax on inherited JSL!

No, not really. So what do we mean by JSL inheritance? Basically it means being able to include JSL from other files into the one being processed. You'd want to do this if you have standard steps used across a lot of different jobs and you don't want to maintain numerous copies. Those with a z/OS background will recognize this as a JCL PROC.

In its most simple form, we'd introduce some new syntax into JSL to allow you to specify the name of another JSL file to go fetch and include as if it was found right here in the JSL file being read. Right away this raises some questions.

Where would we go look for such a file? Would it have to be in the same location as the original JSL file was found or could you specify a path to find it? Would there be some sort of standard 'search' path we'd look through (like a JCL PROCLIB DD..if that helps anybody).

Presumably the included JSL would just be a set of step elements. Would the included segment have an id value so that flow control could just branch to whatever was found first inside the included file or would you need to know an id from inside it in order to flow into the steps? How do you leave the included segment? Would some flow control inside that file need to know an id of an element outside of it? Or would it just run the included segment until it ran out of steps and then…do something else? Would the include element specify where to go when it ended? Have some sort of 'decider' like capability?

Could you include a JSL file from within an included JSL file? If so, how many layers in can you get? Can you have a circular include (hopefully not, but what happens if you try)? Can you have conditionally included JSL?

What about parameters? Can you specify parameter name/value pairs to be passed into included JSL? Would included JSL be able to use parameters known outside the included segment? Do you have to include a full step, or could you, for example, specify a chunk step but include the reader/writer/processor?

At a high level it seems like you'd just go read the included file when you're reading the initial JSL and then treat it all like one big JSL file, but I think in practice it gets a lot more complicated. Feels like it could be really powerful, but there are a lot of rules to be worked out.

Jakarta Batch Issue 109 – Job Definition API

The issue can be found here.

The original specification requires the job to be defined through an XML file included as part of the application package. The specific XML syntax is called Job Specification Language, or JSL. The Liberty implementation allows you to provide the JSL as part of the payload on a REST API to start a job, but it is still JSL and has to exist somewhere. The idea behind this proposal is to be able to define the job on the fly, from some sort of API.

Of course, one option would be to just let you use one of the various APIs (JAX-P, etc etc) that allow you to programmatically build an XML document. The Job Operator API could be updated to allow you to pass in one of those document types as a different signature of the start method. Instead of specifying the name of the JSL file, you would just provide the JSL directly as an object.

Handy, but still basically just JSL. We could take this a step further and provide an API (or perhaps a set of APIs) that allow you to define the job more directly, without going through XML definition. Of course, it would be fairly similar to creating an XML document. Presumably you'd call some API to create a job, then another to create a step within the job (like creating an element within an XML document). But you'd have actual step objects to manipulate.

I'm honestly not sure that having specific APIs like this really gets you very much if you're essentially just building XML anyway. But, if we know the job is going to run right here in the same JVM that is defining the job, well then we could add all sorts of things that just can't be done in XML. Instead of having a property for some batch artifact value that's a string in XML, you could have the value be an entire object. All sorts of possibilities open up if you don't have to flatten everything into text.

But, of course, this then precludes the ability to run the job in a different JVM. Although I suppose you could require that any objects you used for this sort of thing be serializable. You wouldn't typically try to type out a serialized object into XML yourself, but a generated document (or a 'job' object) could be easily defined to accept one.

I'm sure there's lots of other clever possibilities here that I'm overlooking. The whole idea seems really interesting to me, but I think it also opens up a lot of possibilities for confusion and complexity, so we'd have to be careful not to define an entirely new specification with all different capabilities.. if that's what we want, perhaps a totally separate specification is called for, instead of complicating this one. Maybe.

Thoughts?

Jakarta Batch Extras – Apache BatchEE and JBeret

There are a lot more open Jakarta Batch issues, but I've run out of the ones that I thought were interesting enough to write about. If I skipped your favorite one, feel free to post something here about it.

What I'd like to do now is start looking at some of the readers, writers, processors, and batchlets that are available for you to use. The ones I'll be talking about are available from Apache BatchEE and JBeret. I'm sure there are others in other places, but these are the ones I felt like writing about.

Apache BatchEE started out as a fork of the reference implementation of JSR-352 written by IBM as part of creating the original specification and TCK. As we've noted earlier, Open Liberty is also derived from the same reference implementation.

Like Liberty, the BatchEE implementation added some things. Some of these are built right into the implementation, like the ability to have @StepScoped and @JobScoped objects. That's another idea that has been kicked around for inclusion in the specification.

They also included global step and job listeners. You can define these to get control for jobs/steps that run in a given environment without needing to include them in the application itself. This is somewhat like JES Exits on z/OS, if that helps…

They also added a JMX wrapper around the JobOperator interface. This is kind of like the REST interface that Liberty has provided for the same purpose.

The BatchEE implementation also includes a GUI to help you perform Job Operator functions. This is similar to the Batch tool that is part of the Liberty Admin Center. The BatchEE GUI rests on the JMX wrapper much as the Liberty Batch tool rests on the (ahem) REST interface.

JBeret is part of the WildFly Java EE implementation and handles the Java Batch part. They also have a REST API and a GUI.

Both both JBeret and BatchEE have a set of Readers, Writers, Processors, and Batchlets (JBeret calls them 'support' and BatchEE calls them 'extensions'). These should be able to be used (perhaps with a bit of touching up) in any JSR-352/Jakarta Batch implementation (the wonders of an open specification).

In the coming weeks I'll be taking a closer look at some of these and how they work.

Apache BatchEE – ChainProcessor/ChainBatchlet

The first BatchEE extension we'll look at is the ChainProcessor and ChainBatchlet. These are located in the extensions/extras folder and are in the org.apache.batchee.extras.chain package.

As you'd expect, ChainProcessor implements the standard ItemProcessor interface and ChainBatchlet implements the standard Batchlet interface. Which means the processor has a processItem method and the batchlet has an invoke method.

Before we get into what they do, we also need to note that both classes Extend the locally defined Chain class. Because they extend that class, it can have batch properties injected into it (if it was just some other object the properties would have to be injected into both the processor and batchlet – this keeps it all in the common Chain code). Let's look at what Chain does.

It actually gets four properties injected, but we'll focus on just one: chain. This is (by default) a comma separated list of classes that implement ItemProcessor or Batchlet that are to be called, in the order specified. The code parses the list apart and new's up an instance of each one. As we've noted before, the Processor and Batchlet interfaces don't have any sort of 'open' method to do initialization and batch properties don't get injected until after the constructor has run, so all this just has to happen as part of the initial invocation. Doesn't really matter for a batchlet, but the processor will be called a lot and we have to have conditional logic around the initialization so we only do it once.

Anyway, after we've found and created all the objects you specified in the chain list, their process or invoke methods are called, in the order you specified them.

What about the results? An ItemProcessor is expected to return some sort of result object that will be given to the ItemWriter. In the case of a ChainProcessor that's the returned value from the last ItemProcessor in the list. So order matters. Each processor implementation will be given the same object provided by the ItemReader, but only the returned result of the last processor makes it to the writer.

In the case of a Batchlet, the process method is expected to return a string which becomes the exit status for the step, so the results of the last Batchlet implementation called is cast to a String and returned.

What happens if an exception is thrown? As written there's no catch logic so any exception thrown by any processor/batchlet implementation will just fail the step right there. You'd want to catch and handle any likely exceptions inside your implementations.

This seems like it could be pretty useful, although you'd have to be careful that any-but-the-last implementation doesn't expect its returned results to be used.

Apache BatchEE – FlatFileItemReader

Lots of batch jobs read records from a plain text flat file. There might be some structure to the records using a separator like a comma or a space or some other character. Or it might throw back to the punch card days and be organized by column. Or perhaps some fields are variable length and preceded by a length. All these possibilities made me wonder how the BatchEE folks could produce a common flat file ItemReader.

Well, what they realized is that no matter what the structure of the record there are some common things you need to do. The first of these is just keeping track of what record you are on. To do this they actually extend another BatchEE class called CountedReader. The CountedReader tracks the record being processed and provides the count as checkpoint information. On a restart it handles positioning itself in the flat file at the next record to process based on the checkpoint information. This is common function you'd have to provide no matter what kind of flat file you're processing, so why not have a standard implementation to extend?

But the CountedReader doesn't actually open the file or read records. It assumes that will be handled by some extending class that also implements a doRead method to actually read the records. Our FlatFileItemReader does just this. And that is where the injected properties come in that tell us what file to open.

FlatFileItemReader also allows you to specify, via injected parameter, a column one character that indicates a comment row. Any record read that begins with this character will just be skipped. The default is the pound symbol '#' (aka hash, number, or octothorp (yeah…really..new one for me)).

All this gets us the basics of managing and reading through a flat file as part of a Jakarta Batch job, but what about the format of the file? By default the FlatFileItemReader will just return whatever it reads as a String. That's probably fine if your processor can deal with it that way.

If you need to do some parsing of the record, you can implement your own LineMapper (another BatchEE interface) with a map method and the FlatFileItemReader will call it after reading each record. The map method gets passed the String that is the record read, along with the line number in case that helps somehow. Your map function can then parse away and return any Object it likes for the processor to handle.

Which means that all you really need to do to read a flat file in any format is write the bit of code that pulls the record apart. The BatchEE code handles all the boring stuff for you.

Apache BatchEE – FlatFileItemWriter

This seems pretty easy given the FlatFileItemReader. This is just an ItemWriter that allows you to inject properties for the file path/name, a line separator, and an encoding to use. It handles the basic file stuff as you'd expect, incorporating the separator and encoding along the way.

How does it convert the Objects provided from the reader/processor into something it can write into a flat file? The assumption is that toString( ) will probably get what you want. The Object is coming from your application code (probably the ItemProcessor) and so you could implement your own toString( ) that returned whatever you wanted the writer to put in the file.

If somehow that doesn't work for you, you could always just extend the FlatFileItemWriter with your own implementation because the toString is done in a method called preWrite that you could override to do something more complex that toString if you have to.

Well what about restart? The FlatFileItemReader handled checkpoint data and repositioning itself within the file at the record it was at when the last checkpoint committed. What does the writer do? Well, it will position itself at the location of the last checkpointed write, but it isn't really clear to me that doing that will just overwrite anything written to the file after the last checkpoint.

The implementation pulls in another BatchEE extension called a TransactionalWriter. I spent a little time poking around in this and frankly I'm skeptical. Maybe it works. I didn't spend that much time on it. But I'm highly suspicious of anything that tries to make a plain file transactional. There's some stuff in there with JTA User Transactions and maybe that somehow takes care of it.

Generally any Java Batch job that is writing output to a basic file is going to have to do some work to sort things out when there are failures and restarts because normal file I/O just doesn't roll back.

To be clear, I'm not saying I know this doesn't handle rollbacks properly…I'm just skeptical. I would be delighted to be convinced it handles restart/rollback situations correctly.

Jakarta Batch Extras – Reading CSV Files

As we discussed earlier, you could use the FlatFileItemReader to read a file and you can get control to parse it apart. If you are reading a CSV file (Comma Separated Values) you could extend what the Flat File reader does to handle those. In fact there are several existing things you could exploit instead of writing a parser yourself.

And that's what both the JBeret and BatchEE folks did. JBeret offers a JacksonCsvItemReader and also a CsvItemReader. BatchEE has a JSefaCsvReader. Now before I peek inside, I just want to remind you that there's nothing in those that are specific to JBeret or BatchEE (at least that I spotted). You should be able to use those pretty easily with any JSR-352 implementation. So if you tell your team lead that you're planning to use a JBeret item reader in your batch application, don't let them tell you no because you're not using JBeret.

Ok, so digging around inside all three of these reveals that they all pretty much work the same way, which you'd expect. Like the FlatFileItemReader, they all build on top of base classes that implement some of the drudgery of managing your position in the file for restarts and that sort of thing.

And, ultimately, they all read and parse records out of the file. The big differences are essentially in which framework they use to do the parsing.

The CsvItemReader uses SuperCSV which you can find easily with a quick internet search. How 'super' is it? Beats me, but it sure looks pretty easy to use. And the code has a bunch of injected values you can use to customize how it behaves.

Staying with JBeret, there's also the JacksonCsvItemReader which, as you might expect, uses the Jackson parser. Also easy to find information on. Jackson is, I think, more known for being a JSON parser (and BatchEE has a Jackson JSON Item Reader), but it supports other formats including CSV.

And finally, there's the BatchEE JSefaCsvReader. No surprise, it uses the JSefa parser (Java Simple Exchange Format API) to handle reading the csv file. JSefa also handles a bunch of other formats and you can read more about that easily enough.

So which to use? Well, there are capability differences and I'm sure some benchmarking would show some performance differences that probably vary depending on your data. The main thing is that the heavy lifting has been done for you already. Give 'em all a try and see which one works best for you.

By the way, I'm not providing links to any of this stuff because I don't want to suggest I'm endorsing anything. I'm trying to remain neutral. And it is all very easy to find anyway.

Jakarta Batch Extras – Writing CSV Files

Well, with all those ways to read a CSV file that we talked about last time, surely there must be matching implementations to write a CSV file (and stop calling me Shirley…).

Why yes there are. BatchEE provides a JSefaCsvWriter and JBeret provides both a CsvItemWriter and a JacksonCsvItemWriter.

The JSefaCsvWriter is built on a more basic JSefaWriter. The JacksonCsvItemWriter is built on a JacksonCsvItemReaderWriter, and the CsvItemWriter is built on the CsvItemReaderWriter. Those last two are shared with the Jackson and Csv Readers we talked about last time.

Which means that these all work pretty much the same as the readers did. Each one makes use of the underlying library to do the conversion into a CSV format. And in each case it is going to map the attributes of the Object passed into the writer into columns of data.

I thought the header mechanism was interesting. You can, of course, just write CSV data without a first line of column headers. But it is generally more helpful to have a header. The CsvItemWriter allows you to specify a ‘header’ batch property which it will dutifully write after it opens the output file. So you’d have to know in the JSL what you want it to be.

Looking at the JacksonCsvItemWriter I didn’t see a way to get a header at all, but maybe I missed it.

The JSefaCsvWriter has a property indicating whether or not to include a header and allows you to specify the header string, like CsvItemWriter. But it can also look at annotations in the Object being mapped for @CsvField and can use the attribute name as the column header. That’s pretty slick.

I should probably point out here that I don’t have any actual practical experience with any of these. I’m just casually browsing through the code and commenting. If you’re really interested in using any of these you should definitely do a little digging. I’m sure there are online resources where you can get less casual information and answers to questions. I just found all this stuff to be pretty interesting and worth a casual stroll through it..

Jakarta Batch Extras – JSON and XML Readers and Writers

I could probably stretch this out into several different posts, but it is beginning to feel a bit repetitive. We've talked so far about flat file readers and writers and CSV readers and writers. Fundamentally we're just reading and writing plain text files with some internal formatting and either parsing the syntax or taking an object and putting its attributes into that syntax.

JSON files and XML files really aren't any different. The mechanics are all the same, it is just a matter of how you handle the format within the file. Ok, XML is a bit more complicated because you don't just have one record of the file as a single thing to read. In fact, as I think about it now, what would you do reading an XML file in as input to a chunk step? I guess you could have an outer element that defined a list of some sort and then inside it a bunch of the same element over and over. You'd parse those inner repeated elements to pull out the attributes and that's the data for your Read results.

But a more complex XML file…? Maybe there's something specific in there you're looking for and you'd just parse along until you found it I guess..

Anyway, back to the topic.. BatchEE and JBeret both have reader and writer implementations that can handle these formats too. There's a STAX based XML reader. There's a Jackson JSON parser, along with a JSON-P parser. There are probably some more in there I missed.

And all of them have a bunch of injected parameters to let you manage and control what it is doing and to help it map the data it is reading into an Object (or map the Object into data being written). Certainly none of these is just going to magically read your particular JSON or XML and produce your intended Object.

But, like a lot of the other implementations we've discussed, these handle all the basic mechanics for you. You just need to figure out what parameters to feed it and what other things you need to implement specific to your data. The boring generic stuff is done.

Jakarta Batch Extras – JBeret ArrayItemReader

We’ve been talking a lot about reading flat files in various formats and essentially you can view the whole file as an array of whatever it is you’re reading. Whether it is CSV data or JSON or some syntax of your own, you’ve got a bunch of things that are basically the same, in a fixed-size set which is pretty much an array. So that got me wondering what the ArrayItemReader does…

It is pretty cool because it reads an array of whatever you want to give it. It is based on the ‘resource’ property that gets injected from the JSL. You can point it to a file or you can just give it some inline data enclosed in square brackets (those are important – that’s how it knows it isn’t a file).

Then the ArrayItemReader will just crack that open and start reading things from it. By default it assumes every entry is a String and will return them to you that way. But you can have it read some other primitive type or read some object type of your own.

Probably the best way to understand it is to look at the examples in the prolog for the source, which I’m going to shamelessly steal and put here..

```
<property name="resource" value='["a", "b", "c"]'/>
```

That array of 3 items will be read as strings and you’ll make three passes through the chunk loop. But what if you had numbers? Something like this:

```
<property name="resource" value="[1, 2, 3]"/>
<property name="beanType" value="java.lang.Integer"/>
```

Then the ArrayItemReader will use the beanType value to figure out these are integers and read them (and pass them to the ItemProcessor as) Integers.

Or you could have some JSON (and here I’m simplifying the example in the code):

```
<property name="beanType" value="org.jberet.support.io.Movie"/>
 <property name="resource" value='[
 {"rank" : 1, "title" : "Number One", "opn" : "2017-01-01"},
 {"rank" : 2, "title" : "Number Two", "opn" : "2017-02-02"}
]>
```

Here the application object (Movie) will be used to map the JSON array entries (just two here). And if you don’t like instream data, drop the square brackets and put the data in a file:

```
<property name="resource" value="movies-2012.json"/>
<property name="beanType" value="org.jberet.support.io.Movie"/>
```

Jakarta Batch Extras – BeanIO Reader and Writer

There are just lots of different ways to process various formats of flat files, and the JBeret and BatchEE folks seem to have created readers and writers to use them all. This week I'll take a quick look at the ones that use BeanIO.

So what is it? BeanIO (according to the web site) is, "an open source Java framework for marshalling and unmarshalling Java beans from a flat file, stream, or simple String object." Well that sounds a lot like some other things we've seen here. From poking around in their doc it sounds like you can create an XML document that describes the structure of the file you're reading, or you can just annotate the Java class you want it read into, and it mostly takes care of the rest.

That means the reader and writer take care of the usual sorts of things. First off is the management of position within the file and handling checkpoint data received on a retry or restart. It also handles all the interaction with BeanIO. Of course there are things you can give it, through injected parameters from the JSL, to configure BeanIO so it knows what you want.

All that gets passed to BeanIO as part of the setup (open) and then it just slogs its way through the file, letting BeanIO do all the parsing and whatnot. It feels like you could probably just use this as-is by setting up the right stuff for the file format.

Well, by now you're probably wondering if you should use this or one of the other reader/writer implementations we've looked at. Good question. As I've said before, there are probably some performance differences in all these that will no doubt vary with your actual data and structure, so do some testing. There are also a lot of subtle differences in the kinds of things they deal with. If you've got nice normal looking stuff with no quirky syntax, then you have more choices, but I suspect there are some odd cases that some of these handle better than others. On the other hand, more configuration options generally slows things down, so simple is sometimes better…and easier to maintain.

I tend to look at all these different reader/writer implementations from a different perspective…it isn't so much..I have a flat file in some format and I need to read/write it. I think it is more, I was told to use this tool to process our file in this format and I need a reader/writer to do that…and JBeret and BatchEE provide a lot of them. Or at least a pretty good starting point.

If you are coming in cold and need to figure out which way is best to read/write your particular file format…you have a lot of options and you're going to need to experiment a bit.

Jakarta Batch Extras – Reading and Writing Excel Files

More specialized readers and writers for quirky file formats? Seriously? I swear this is probably the last one. Not that there aren't more, I'm just running out of ones I find interesting enough to talk about (which is, perhaps, well past what you're interested in reading about).

So you've got some spreadsheet that you need to process with your batch job. Wouldn't it be easier to just export the thing into a .csv file and read that? Or maybe you've got to look at the results of a batch job in Excel. Wouldn't it be easier to just write a .csv file and import it?

Well, maybe. But possibly the spreadsheet you have to process gets emailed (or sent in electronically some other way) and converting it to a .csv would require some manual intervention. Nobody wants to be the person who has to go do the dumb format conversion so the stupid batch job can read the file.

Or maybe the result of the job gets emailed to some executive and you don't want to have to explain to them how to import a .csv into their spreadsheet application. They want a .xlsx file because that's what spreadsheets are (no offense to any executives reading this of course).

Or…maybe…you're trying to write a job that does an automatic conversion of a .csv file to/from a spreadsheet. There's no 'processing' involved but just a straight up format conversion step.

For all of that sort of thing you'll need an Excel reader or writer. And, these things being what they are, you'll need one that can handle the various incompatible format changes that seem to come along now and then. Which means this isn't necessarily as simple as it might sound..and it probably requires some ongoing upkeep over time.

Of course, this….. isn't being done with specific knowledge of the file format buried right there in the reader/writer. They've exploited Apache POI which exists to handle Office Open XML standards and that's where the magic happens.

Alright…I'm done with flat file readers and writers, at least for a while. Next week, something new-ish.

Oh, and of course "Excel" is a trademark of Microsoft. Lest we forget.

Jakarta Batch Extras – NoOp and Mock Writers

A few weeks ago we talked about a proposed update to the batch specification that would make the writer optional. Sometimes you just don't need one. In the absence of that update, what do you do? Well, you create a dummy or no-op writer that implements the interface but doesn't do anything. If this sort of thing happens to you a lot, you'll hopefully not go creating a dozen no-op writers that are all pretty much the same. Not that it will require a lot of maintenance, of course, but it just feels sloppy to have a bunch of these useless things lying around. And if you're going to have your own sort of standard no-op writer, why not just use one somebody else wrote that exists in open source?

And thus, the Apache BatchEE NoopItemWriter. It implements the ItemWriter interface, but does absolutely nothing. Enjoy.

Well, ok, but what if you don't really NEED to write the results of the processing anywhere, but you might like to know what they were anyway. Maybe just during testing. The ItemProcessor does all the stuff that needs doing, but it does return something. Maybe some indication of how processing went or something like that. Not part of the results and not something you need to keep around when the job is in production, but maybe a nice verification things are going as planned. And maybe something to tuck in that you could turn on dynamically in case there are problems (what? Problems in a production batch job? Say it ain't so!)

Well, for that sort of thing you should have a look at the MockItemWriter provided by JBeret. By default it does a println of a toString of the objects to write. But you can disable that from an injected parameter so it does nothing at all.

You can also configure it to write the objects into a specified file (again using toString on the object). That may or may not be better than writing into the log depending on your environment.

And finally, you can provide a class and a List attribute of that class and MockItemWriter will add the objects to write into the list. You could have a listener that gets control after each chunk or at the end of the step and looks through the contents of the list and flags anything that looks odd.

Since all this behavior is optional, you can have the job set up to do none of this by default, but easily use job parameters to turn it on if you need it.

And there you have it…a blog post about code that does nothing.

JBeret – OsCommandBatchlet

This is pretty simple…a batchlet that uses the Java ProcessBuilder to run a set of commands that get injected from the JSL. You can do things you'd expect like setting environment variables before the command runs and defining a working directory.

You can probably imagine all sorts of things you might want to do with this. Sure you could issue shell commands, but you could also run a script. And a script can do all kinds of things, maybe even launch another JVM…although I don't know why you'd want to do that.

In any event, you're pretty much just limited by your imagination. You can use this to run nearly anything asynchronous to the Java Batch job.

Ah…so what does that mean? Well, this is going to launch an entire process that will run separately from the JVM running your Java Batch job. Should you wait for it to complete? The OsCommandBatchlet will wait…for a while. You can specify a timeout after which it will just give up.

That's kind of interesting. What happens if you give up? You've got no idea what happened to the running process. Maybe it died and you didn't find out? Or maybe it is hung somehow? Will it start running again and finish? Your job has moved on and probably assumed it just didn't work. Looking a bit closer I see that it does a destroyForcibly( ) on the process when it times out. I wonder how guaranteed that is to work? Ah..there's a waitFor( ) to wait for it to actually finish so you know it isn't still out there running. If there's a chance it doesn't forcibly end, you'd be stuck there.

So there's an aspect of this that makes things complicated. It is very handy to do, but error handling can get a little dicey. Your script could leave marker files around to indicate what it did (or didn't) do. Then after the batchlet says it timed out you could go looking for these little footprints to try and see if there's anything you need to undo or, well, whatever.

The batchlet also generally assumes that whatever you run will return a zero when it works, but there is a parameter you can inject that indicates what a success would look like, in case whatever you are running returns something else to indicate success.

All in all it seems like a pretty handy way to do something quick, but I think I'd be a bit careful in a production job and worry about error cases.

Apache BatchEE – JdbcBatchlet

This is an interesting one. I'm not entirely sure how useful it actually is in the real world, but structurally it lays some groundwork we'll see used in other BatchEE things. Essentially this batchlet allows you to issue a single SQL statement without having to write any code.

Let's start with the connection to the database because we'll use that for other JDBC related things. The batchlet extends another BatchEE class called JdbcConnectionConfiguration. That takes an injected value of the JNDI name for the datasource you want to use. This is the easiest thing to do. If you don't have a JNDI name you can also directly provide a driver class name, a URL, and a userid/password and the connection configuration class will try to use those to get a connection to your datasource. This class doesn't really do very much…just gets a connection to the specified datasource, but it factors out that so it doesn't clutter up the batchlet or, as we'll see later, the reader etc.

Which brings us to the batchlet itself. This too is pretty easy. It uses the connection configuration to get a connection to your datasource, then prepares and executes the SQL statement you injected as a batchlet property. There's nothing clever being done here…it just executes the statement you gave it. Then it closes and commits to tidy up and we're all done. The batchlet's stop method does nothing.

Whatever is returned by the execution of the SQL is returned from the batchlet. This becomes the exit status value for the step. You can use this in the JSL in conditional routing. You might be able to try to insert a row and conditionally go to one step or another depending on whether the insert works or not (maybe a row with that key already exists). Or delete a row and make a decision about what to do next depending on whether it existed or not.

You could probably do some other things like a SELECT COUNT( ) and make a decision based on the number of things found (less than 100, we're pretty caught up so move on, more than 100 and we need to do some processing…or something like that).

When I started writing this I thought it was rather odd that you'd have to basically hard-code the SQL statement which seemed rather limiting to me, but now that I've thought about it some I can see that you could actually do some pretty clever decision making with it…all without actually having to write any code.

Apache BatchEE – JdbcReader

This handy little class combines several things we've talked about already. Essentially it is a standard ItemReader that you can use to pull records out of a database table. If you think about how you'd write something like that, you'll have some obvious questions. How does it know where the database is? How does it know what table? How does it know what rows to select and what columns I need? Once it has a row, how does it produce the Object that gets passed to my ItemProcessor?

If you think back over the past weeks, you'll see we've already got answers to all of that. Just like the JdbcBatchlet, the JdbcReader extends JdbcConnectionConfiguration which handles all the issues with getting a connection to the database using injected values from the JSL.

And just like JdbcBatchlet, the JdbcReader allows you to specify an SQL statement to be executed. This time it will get a result set back that it has to iterate through instead of just returning the result as the exit status. That's all good too.

But how does it know how to convert the results into an Object? Well, it does it almost the same way that the FlatFileItemReader does it. For the flat file the reader would, by default, just return a String that was the contents of the file record. However you had the option of implementing a LineMapper class that would convert that String into an Object.

In the JdbcReader you have to provide a RecordMapper. The mapper will get the result and can map it into any Object it likes, however it likes. The results are added into a linked list maintained by the JdbcReader and that will be returned as 'read' results until it runs out.

This Object would naturally be the actual class expected by your ItemProcessor.

All in all a pretty handy little class that allows you to read records from any JDBC accessible datasource, handling all the gorp while you just worry about mapping the results into your object.

That said, it looks to me as if the reader reads all the rows matching the SQL statement you specify, passes them through the RecordMapper, and then keeps the resulting objects in memory. That might be a problem if your table has a lot of records. Or perhaps I've missed something about how it works.

Apache BatchEE – JdbcWriter

By now this should be pretty obvious. The JdbcWriter extends the JdbcConnectionConfiguration which gets all the injected attributes to establish a connection to your database. Just like the other JDBC batch artifacts.

And you're going to want to specify some SQL statement to do the insert of each row into whatever table it goes into. And just like JdbcReader you get to specify one. No surprises here either.

The only thing left to do is figure out how to convert the Object that your ItemProcessor provided into something that can be inserted into your database table.

The FlatFileWriter had a method you could extend to handle the conversion. For the JdbcReader we had a RecordMapper that mapped the record that was read into an Object. But in this case we want to go the other way. We've got an Object to start with, which is probably why the interface you implement here is called ObjectMapper.

The ObjectMapper gets passed the Object provided by the ItemProcessor and the PreparedStatement that resulted in calling prepareStatement on the database connection with your specified SQL. In your ObjectMapper you get to figure out how to extract data from your processing result and push it into the PreparedStatement so that it can be executed.

One neat little bit to this is that the JdbcWriter calls the ObjectMapper for every Object it was passed (all the results for this chunk of processing) and then it calls executeBatch on the resulting complete PreparedStatement. This will essentially insert all the rows needed, all at once. Generally, this gives you much better performance that doing the inserts one by one.

We've got a bit of a theme going here. All of these different implementations of batch artifacts handle the basics of dealing with whatever datasource is involved while leaving relatively easy ways for you to get in there and fiddle with things where your actual data has to come into play.

Jakarta Batch Extras – Using JPA Readers and Writers

Both JBeret and BatchEE provide JPA based readers and writers. JPA is essentially a layer built on top of JDBC that provides an abstraction between the application code and the code that interacts with the database itself.

This layer of abstraction allows the application to be less concerned about the specific database implementation provided. Anybody that has dealt with several different database products over the years knows that while SQL is a standard, there is a lot of gray in how it is implemented. Things that work just fine with one database will require some tweaking to work with another. Which means you can't just switch your underlying database without having to change the application (or write some complicated application code to handle the differences).

JPA abstracts that away. It worries about the specific syntax differences and lets you just write your application code. Now it is certainly unlikely that one day your management will tell you they are moving all their production databases from one vendor to another, so you probably won't have to go change all your production batch applications overnight. But you might want to write applications that work for different parts of your company that might use different database products and not have multiple versions. Or maybe you want to write applications used by different companies.

Additionally, JPA just makes it easier to deal with an underlying database. You worry less about the details and focus on the application things you need to do.

The readers and writers provided by JBeret and BatchEE don't just magically work for you (nothing every just magically works :-). But they provide the basics into which you can insert the JPA application stuff that you have to write. You give it the name of the named query you need to run, and it handles actually doing that and pulling out the result set.

Like a lot of the readers and writers we've looked at, these, at the very least, provide some sample code to build your own application on top if, even if you can't just use it as-is.

Jakarta Batch Extras – NoSQL databases

What about readers and writers that interact with NoSQL databases (or non-relational databases, or not-only-SQL databases, opinions seem to vary)? For this we're looking at things like MongoDB and Apache Cassandra (there are certainly others). JBeret offers readers and writers to interact with these also.

I don't want to get into a discussion about database preferences and the pros and cons of relational databases vs non-relational. Writers of batch applications frequently don't get any choice. The data is already wherever it is. What you're looking for is a way to get your batch application to read it (or write into it). Maybe your batch job is just a converter from one datasource to another. A lot of batch jobs read flat files in various forms and insert records into databases.

Looking over the code, the two things are quite different. That makes sense as Cassandra and MongoDB are quite different. For the reader with MongoDB you specify criteria for the 'rows' you want and with Cassandra, well, there's CQL statements. But I thought it was interesting that both readers have a 'batch size' parameter you can inject.

The batch size controls how much it pre-reads to get ahead. The results are stored locally to avoid trips back and forth to the database. On the one hand, you want to read ahead a little bit to avoid making a trip to the database for every row. On the other hand, you don't want to store a mountain of data locally. Some of the readers we've looked at earlier appeared to pre-read the entire set of results into memory and then just iterate over it. The batch size configuration allows you to find a good middle ground between needing too many trips across the network against needing a giant heap.

This is going to wrap up our review of readers and writers that interact with files or databases. The next couple of weeks will look at interacting with other sources of data.

Jakarta Batch Extras – Reading and Writing Messages

Ah batch that involves messaging. We talked about this months ago with the sample pipeline job I wrote. In that case I had a split flow with the writer of one flow feeding the reader of another flow through a message queue. There's a lot you can do with this.

Well, first I should go through what JBeret and BatchEE provide since that's what brought up the topic. There are readers and writers that use JMS, but also ones that work with Apache Kafka and Apache Artemis, and Apache Camel. You can go look those up, but they are all essentially messaging based (argue in the comments, of course, but I'm at the waving-my-hands-around level).

Messaging writers are pretty cool. Basically, you're converting whatever you read (flat file, database rows, etc) into a message to be processed by something else. The job step is essentially a protocol converter. It can also be a way to get some concurrency. Rather than have the job process record by record, convert them into messages to be processed as message-driven-beans in a multi-threaded application server.

Messaging readers raise an interesting question. When is it done? It is pretty easy to dream up a job that just reads messages from a queue forever, pushing them through the processor as they turn up. But what you've really done is created a server and not a batch job. Do what you want, of course, but I think this sort of thing should be run periodically until the queue dries up and then terminate. So every day (or ever hour, etc) the job runs and handles all the messages that have turned up since the last time it ran. Kind of a "your update will be processed within the next 24 hours" kind of thing.

As you might guess, my favorite use for messaging based batch jobs is for pipelining. Multiple jobs, maybe running concurrently, or multiple flows in a split in a single job, with data flowing between the jobs/flows. One flow does some processing, feeds results to the next flow through a message queue, which does a bit more and passes it along.

Rather than run one thing then the next, it all kind of happens at once. Cleanup if bad things happen can get a bit tricky, but it can be handled.

Not that this is a new idea…. I remember people working on BatchPipes on z/OS decades ago.

Jakarta Batch Extras – REST Readers and Writers

I suppose this should have been obvious, but I have to admit when I saw this in the JBeret set of readers and writers I thought, “Wow…that would be cool”. So, as with messaging, let’s consider the writer first.

Basically you’re going to give the writer a URL and it is going to do a PUT or a POST with whatever object payload you give it to write. It is just going to drive this same REST interface over and over as the job runs, pushing data at it to do whatever it does. This seems like a really great way to integrate something at least kind of modern with something like batch that feels, well, less modern, no matter how you write it? I really like this. You’ve got some nifty OLTP REST thing that gets used by your mobile users and you drive that same interface from your batch program. Of course this might subject it to a short, but intense load it doesn’t normally get. You might confuse the heck out of your dynamic scaling…one minute hardly anything, the next you’ve got 1000 requests/second, and then nothing when the job ends.

Alright, but what about the reader? Well, that’s just going to drive a GET against the URL you give it and turn the response into the read results. But think about that for a moment. There’s an assumption that each call returns a different result. So either the client is passing in some information that changes (like incrementing through record indexes or a list of customer identifiers) and getting back the corresponding result, or the REST provider has some state. If you don’t tell it what record you want, there’s an assumption that every time you call you get back a different result to be processed.

Now maybe the GET operation does update something to indicate the result it passed back has been ‘consumed’ and it won’t give it back again. Somehow you’d have to reset that to run the job again. Unless, like messages, the returned result is actually consumed when it is returned and doesn’t exist to be given out again. But now you’ve got some worries about job failures. What if the job died before it finished processing the record and now it is gone? Well, maybe it is ok, but maybe it isn’t.

It seems like it would make more sense for the client to pass in some indication of what it wants back. The REST ItemReader from JBeret has a readerPosition variable (an int) that gets passed to the REST interface to indicate the record you want. And you can control where it starts, but it always increments by one.

I noticed that you can also get the reader to do a DELETE instead of a GET. That seems kind of weird for a reader, but I suppose maybe that fits with the give-me-the-next-one model where the server forgets the item when it has returned it.

So some quirky stuff here, but all in all I think it is pretty cool.

Java Batch How-To: Introduction

We've been looking at some pretty practical aspects of writing Java Batch applications using the JSR-352/Jakarta Batch specification.

That got me to thinking that perhaps I should officially introduce a new topic. I'm calling this the JSR-352 "How To" series. Each week I'm going to pick some aspect of the programming model and cobble together some little sample that shows how to make use of it. These samples are going to be hokey little things that don't actually do anything useful. But, unlike a lot of samples, I'm not going to try to show you a bunch of different things all at once. Each sample will be just enough to illustrate one point.

Well, ok, that sounds like a fine thing to have in a repository in github, but it hardly seems interesting as a blog. Surely I'm not just going to paste a bunch of code in here and be done? (Stop calling me Shirley…).

Well no. My hope is that I'll have actual code samples in a github repository, but the blog will have no actual code and instead will be a discussion of what I did in the sample. Some things may have different possible approaches, or perhaps warnings about easy to make mistakes. We'll see how it goes.

So next week we'll get started with our first "How To" topic and hopefully I'll have something interesting to say about it here, and a link to the github repository where you can find the code I'm talking about.

Java Batch How-To: Using Exit Status to Influence Job Flow

We're going to start out our series of 'how to' posts with a simple one. How do you set the exit status from a step and use that to influence the flow of control within the job?

If you're coming from a mainframe JCL-based background, this is a pretty familiar concept. You set the return code from the step and that turns into a condition code in the JCL. Flow control statements in JCL can be used to direct the job to whatever step is next based on the value of the return code / condition code.

In Java Batch we have a similar concept in that the program can set an exit status for the step and that value can be used to influence the flow of control within a job. In our case the exit status is NOT the return value from the step. For a batchlet step the returned String becomes the exit status, but for a chunk step, well, which batch artifact would set the return value? Instead the step has a context that can be accessed from any artifact that runs as part of the step, including any listeners you might define. The exit status is an attribute of the `StepContext` and there is, naturally, a setter method you can use to set a value.

Of course, being able to set the exit status from different places means the setting of the value might not all be contained within a single part. You might set it normally from the batchlet, but have a step-end listener that might also set a value.

This is important because you need to coordinate between what value the application sets and what values the JSL is looking for to do flow control. Which brings us to the next thing that might feel a little odd. The exit status value is a string. If you are used to numeric return codes and condition codes you might be comfortable setting an exit status value to 0, 4, 8, or 12. You can still do that, but the value will really be a string that is "0", "4", "8", or "12". It can just as easily be "OK", "Warning", "Error", or "Failure".

It doesn't really matter, but the value you choose has to match the value specified in the flow control statements in the JSL. If your code sets the exit status to "OK" in the code, but the JSL is looking for "ok" it won't match.

So what does the code actually look like? In github have a look at the SetExitStatusOk batchlet implementation. It does nothing but set the exit status to "OK". The ExitStatusFlowControl JSL looks for that value and passes control of the job to Step1. If the value isn't "OK" then no next step is set and the job ends.

The sample parts are here: https://github.com/follisd/batch-samples

Java Batch How-To: Passing Data Between Steps

A common problem in multi-step jobs is figuring out how to pass data from one step to another. In cases where one step does some processing on a file and a subsequent step does some processing on the results, then you are just passing files around. File naming becomes the issue then. Should they files be permanent files in which case you can name them something appropriate, or are they just temporary files as the job runs that you'll throw away later? If they are temporary, do can you put them somewhere separate from other instances of this job or somehow name them in a way that avoids collisions?

A database is, of course, another good place to keep data between steps. Whether the data is permanent or transient it is very reliable and might be necessary to allow the job to restart if it fails part way through. The `JobContext` object has persistent state data that you can use for this purpose if it isn't a whole table full of information you need to pass between steps.

Speaking of the `JobContext`, the job also has transient data that you can set. If you don't need the persistence, using the transient data methods can be pretty handy. Of course the data goes away when the job ends (successfully or not).

Another nice aspect of the transient data is that it doesn't have to be serializable because it isn't going to be written anywhere. All you need is an Object so basically anything you need. But there are some tricky bits to using it.

First of all, there is only one transient data object for the job. If Step1 wants to use it to pass something to Step4, then Step2 can't also use it to pass something to Step5. If you plan to use it a lot then you need to create some aggregating Object that contains the objects for the various steps to use. Not a big deal, but it requires some coordination across the job which may make it difficult to just click together different bits from different jobs to make a new one.

Another tricky bit to use of the transient data is the possibility of multiple `JobContext` objects for the job. If you use split/flows or partitions each thread can end up with its own `JobContext` and thus separate transient data.

But in the simple case where the batchlet that is Step1 just needs to pass some stuff to the batchlet that is Step2, it is pretty easy. And that's the case our sample shows. Our JSL file is JobContextTransient.xml and it just runs two batchlets one after the other. The BatchletTransientSender is the first step that shows setting the transient user data to a simple string. The BatchletTransientReceiver is the second step that shows fetching the string from the `JobContext`.

The sample parts are here: https://github.com/follisd/batch-samples

Java Batch How-To: Setting the Job Exit Status

Setting the exit status for a job is, technically, a pretty easy thing to do. You just inject the `JobContext` and call the `setExitStatus` method passing whatever the status you want to set. No problem.

Well, we've talked earlier about how the exit status is just a string and you can set it to anything you want. But this status is going to get carried out to whatever is managing your batch jobs and you want the final status of the job to be something that system can handle. It may be expecting something basic like "0" or "8" to indicate success or failure. It might tolerate "OK" or even "ok" or "Ok". But "Yippee-Kai-Yay" is probably not going to work. Get in touch with whoever owns the thing that's going to control your batch job and find out what sort of thing they are expecting. If you talk them into taking that last one, let me know.

We haven't talked too much about where you set it. In fact, you can set the batch job's exit status from ANY artifact that gets control in the whole life of the job. The `beforeJob` method in a Job Listener (probably the earliest thing that can possibly get control in the job) could decide how this job is going to turn out and set it. Maybe you're just optimistic.

For our sample this time we've got a nice simple one step job where the batchlet sets the job's exit status and then completes. But then we've got a Step Listener whose `afterStep` method gets control and changes the value. And finally we have a Job Listener whose `afterJob` method gets control and changes it again.

At each step along the way we use the `getExitStatus` method to find out what the status was before we changed it. That's handy because you might want to just modify the existing status in some way because this artifact has more or different information than the artifact that already set a value. It is just a String so you can concatenate things together – if that will make sense to whoever or whatever sees it.

If nobody has set the Exit Status at all, the current value is null. That's handy too in case you only want to set a value if nothing else in the job has set it. That's how the batch container knows to set the Exit Status to the Batch Status value at the end of the job if it hasn't been set yet. If you use a Decider (coming up) it will set the Job Exit Status for you too, so watch out for that.

Our sample parts this time are: SettingJobExitStatus.xml, BatchletSetJobStatus.java, StepListenerSetJobStatus.java, and JobListenerSetJobStatus.java. Each artifact reports the job exit status on entry and then tells you what it set it to.

The sample parts are here: https://github.com/follisd/batch-samples

Java Batch How-To: Using Splits and Flows

The split/flow is a JSR-352 construct that allows you to run more than one step (or set of steps) concurrently within a single job. To use it you define a set of one or more steps as a flow by wrapping the step elements in begin-end flow elements. Group together the different sets of steps you want to run concurrently into separate flows. Then wrap all the flows inside begin-end split elements.

When execution hits the split element the execution of the flows inside it will spin off to different threads while the thread executing the split waits for them to complete. Once all the contained flows are complete, the split thread wakes up and continues with whatever is next in the job. You can nest a split within a flow that is within a split. There is no architected limit to the number of times you can nest or to the number of flows within a split, however practical considerations will likely impose some limit, so don't go crazy.

For the purposes of our sample I want to focus on flow control. A split is a job element just like a step and for it to get control it either has to be the first thing in the job or control has to flow to it from some other step using normal flow control mechanisms. That means the split element needs an id attribute to give it a name.

Flows can have a name, but don't need one when they are inside a split (wait, what? When would a flow be outside a split? Hang on…we'll get there). Once a flow gets control inside a split, it works like a little job with the first step inside the flow getting control first and proceeding, just like a job, until no flow control directs it to a next step. You can't have a step within a flow direct control to pass to a step outside the flow.

When all the flows have run out of steps to execute (no 'next' steps) then control returns to the split which goes to whatever step you've indicated should be next. Like everything else, if there is no next step after the split, that's the end of the job.

Finally, what's this about a flow that isn't in a split? That's perfectly legal. You can wrap a set of steps inside begin-end flow elements if you want to. In that case you'd want the flow element to have an id so that some step in the job could indicate the flow is the next thing to process. When that happens the steps inside the flow process just like they would inside a split, starting with the first one and proceeding until there is no next step indicated. Then the flow ends and control proceeds wherever indicated as the next step after the flow. Even here you can't branch out of the flow. Independent flows like this are handy if you just want to group a set of steps together because they are part of some process. It is almost a sort of procedure, except you can't call it more than once within the job.

Our example, UsingSplitFlow.xml, begins with a batchlet followed by a split with two flows. Each flow has two steps. When the split is done we proceed to another batchlet step, followed by an independent flow with two steps inside. Look closely at where and how next steps are indicated.

The sample parts are here: https://github.com/follisd/batch-samples

Java Batch How-To: Use a Decider After a Split

Last time we talked about the split/flow concept that allows you to run multiple different steps concurrently in the same job. We spent a lot of time focusing on how control is managed between steps in each flow and what happens when all the flows are complete. But then what?

If you recall, we said that you can specify a next element to get control when the split is complete. This is an unconditional flow of control. A lot of different things might have happened in the flows that are within the split. There's probably some sort of complex decision that needs to be made to figure out what to do next. A decision that needs to consider what happened in every flow, all at once. For that we have….a decider.

If the split indicates the next thing to get control is an element called a decider then the batch container will give control to an implementation of the Decider interface. Technically you can have a decision element anywhere you can have a step, but it is most interesting right after a split.

The decider has one method called decide. It gets passed an array consisting of the `StepExecution` objects for the final step in each flow contained in the split. The `StepExecution` has methods that allow you access to information about the step, but the most interesting one in this case is the exit status value for the step.

The idea is that each flow will have a final step that will somehow summarize the results of the flow into a String that gets set as the exit status for that step. Then the decider can look at the final result of each flow and reach some sort of conclusion about where to go next. You can also get the step name in case that helps you interpret the exit status.

The return value from the decide method becomes the exit status used for conditional flow control in the JSL. That means part of the decider element in the JSL will include 'on' exit status value 'to' some other step for the different possible values returned from the decider. Coordination between the values in the code and the JSL is required for this to work.

You might be tempted to try to use the `StepContext` to set an exit status for the decider, but it turns out a decider doesn't actually have a `StepContext` because it isn't really a step.

Another interesting consequence of using a decider is that the return value from the decide method becomes the current value for the Job Exit Status. A couple of weeks ago we talked about setting that and here's another way it can get set. Be careful about this if your logic to set the Job Exit Status depends on code that makes sure it isn't already set. A decider might set it for you.

Our sample code is a simple split with three flows. Each flow uses a new Batchlet called BatchletSetExitStatusToParm which allows us to force the Step Exit Status for each flow to a different value without having three different batchlets. Just a shortcut because it is a sample that doesn't do anything real. The decider is called SampleDecider. The JSL for the job is in UsingADecider.xml. Have a look…steal some code. The sample parts are here: https://github.com/follisd/batch-samples

Java Batch How-To: Using a Checkpoint Algorithm

A chunk step reads and process records until a checkpoint is reached. Then the writer is called and the transaction that wraps the whole thing is committed. You can checkpoint after some configurable count of records have been read and processed. Or you can checkpoint after some fixed amount of time. Or you can get a lot more control by writing your own checkpoint algorithm.

You configure a checkpoint algorithm in two steps. The first part is to add an attribute to the chunk element that indicates the checkpoint-policy is custom. That causes the batch container to ignore the count and time checkpoint configuration and look for a checkpoint-algorithm element inside the chunk. That points to your implementation of the `CheckpointAlgorithm` interface.

The interface has three parts. The first is the `checkpointTimeout` method. There are probably quite a few reasons why you might want to set a different timeout value for every chunk you process. I tend to think of this is a sort of backstop to the chunk to make sure it ends in case my code making checkpoint decisions has a bug (a backstop is an America baseball expression. It is there to protect the spectators from wild pitches and other stray fast-moving balls).

The second part are the two methods that get control at the beginning and end of the chunk. They don’t have anything to do with controlling the chunk – there’s no returned value. But it is a chance to know when the chunk begins and ends (and thus how long it took).

The final part is where the action is. The aptly named `isReadyToCheckpoint` gets control after every read/process cycle and decides if this pass through the loop is the right time to checkpoint or not. Based on whatever criteria you like.

For our example, I decided to try to implement my own time-based checkpoint algorithm. Using timestamps collected at the beginning and end of the chunk, plus how long each pass through the read/process cycle takes, I try to determine if the step should take one more pass or go ahead and checkpoint now.

To make my life difficult, the amount of time spent in the reader, processor, and writer varies somewhat randomly up to half-a-second as I try to checkpoint every five seconds. The `endCheckpoint` method reports how many iterations we made and how much I missed the goal by.

Go have a look. The code is in SampleCheckpointAlgorithm.java for JSL in UsingCheckpointAlgorithm.xml.

The sample parts are here: https://github.com/follisd/batch-samples

Java Batch How-To: Using Skippable Exceptions to Handle Bad Records

As your chunk step hums along, reading records, processing them, and writing results, bad things are going to happen. A likely bad thing is some record being read that has something wrong with it. Or some of the processing for a particular record goes badly – maybe you can't find the account information for this account number. What to do? Well, your application is probably going to throw an exception. Then what happens? Well, if you do nothing the exception will blow back to the batch container and the job will fail. If it is just one bad record, you probably don't want that.

You could, of course, catch the exception in your application code and handle it yourself. That's not a bad idea. If the reader has a problem with the record it read, you could just go read another one and hope for better results. At some point you might get enough bad records that something is probably wrong beyond a few bad records and you want to just fail the job anyway. If the reader and the processor can both have problems with individual records, you'd want to coordinate counting bad records between them so you know when to give up.

You might want some code to keep track of which records are bad so somebody can go clean things up. Log a message somewhere or maybe put a message on a queue or maybe even send an email to somebody. More code to write and coordinate between the pieces.

Or you could just let the JSR-352 framework handle a lot of this for you. The first thing to do is define the exception you're going to throw for a bad record to the JSL as a skippable exception. This will prevent the batch container from failing the job when it sees the exception. It will just call the reader and start another pass through the chunk loop as if nothing happened.

You can also define a skip limit for the step to stop the job if there are too many bad records that you want to skip. The batch container will keep track for you and automatically fail the job if you exceed the limit. The limit defines how many are ok to have, so if you set it to 3 the job will fail on the fourth skippable exception.

You can create a skip listener to get control when your reader, processor, or writer throws an exception you indicated was skippable. The listener gets passed the exception your application threw. If you put enough information inside the exception, the listener can take whatever action you decided on to get this record cleaned up.

Our sample starts with the JSL in UsingSkippableExceptions.xml. You can see the skip listener defined there, as well as the skip limit for the chunk (set to 3 so it hits). Our reader, RandomIntegerReader, generates random integers as 'read' values ranging from 0 through 110. The processor, RandomIntegerProcessor, checks for values greater than 100 and throws an application exception, SampleSkippableException, after putting the bad value inside the exception. The RandomIntegerWriter just logs the values that make it through. Finally, the SampleSkipProcessListener gets control for bad values and logs the value from the exception.

The sample parts are here: https://github.com/follisd/batch-samples

Java Batch How-To: Use a PartitionMapper – Part 1

You've got a lot of data to process and you've decided to try to speed things up by using a partitioned step. Partitioned steps allow you to run multiple concurrent copies of the same step, on different threads. If the amount of data is constant or rarely changes, then you might statically partition by including partitioning instructions directly in the JSL of the job. If you want to partition by US states, for example, you're pretty safe with static partitioning in the JSL (trivia – Hawaii became a state in 1959 and is the most recent state added).

But if your data source can change, then you'll need to write a partition mapper to break things up on the fly. The first step in doing that is to figure out how much data you're working with. You might do a SELECT COUNT(*) from a database to find out how many rows you'll have to process. Or maybe you can do something clever with file size. In our sample, we're going to be processing all the files in a specified directory, so we'll just go get the list of files in that directory and the length of the list is our count.

Once you know how much data you have, you need to figure out how to split it up. In our case, the simplest thing to do is just have one partition per file we found. But suppose we find thousands of files? We probably don't want thousands of partitions. If we're going to assign multiple files to be processed by each partition, then the code in the step is going to have to expect that and not just one file to each instance of the step that runs.

What if you are reading from a single flat file? The first partition will start at the beginning of the file, but other partitions might have to read through the early records until they get to their starting position. It might be faster to use a utility to split the file into partitions and then have each batch partition instance read its own file from the beginning. Maybe.

On the other hand, if you are processing records from a database you can just assign each partition to process a range of data using ranges of the primary key value to split things up. But you still have to make a decision about how many rows per partition and thus how many partitions.

How do you decide? Partitioning is most useful when there is no contention between the different instances of the step. Another factor to consider, though, is how many instances will be allowed to run at once. You might create 100 partitions, but only allow 10 to run at one time. That's another thing you can set as part of the partition map. The batch container won't necessarily run that many at once, but it won't run more than that.

In the end you'll likely just run some experiments and see how it goes. Remember that your goal is to reduce elapsed time for the job. Contention problems might make it faster to just run the step unpartitioned. System resources might limit how many partitions can reasonably run at once.

I’ve given you a lot of things to think about, but we haven’t talked about how to code this. That’s for next time.

Java Batch How-To: Use a PartitionMapper – Part 2

You've read last week's post and given some serious thought to how you want to partition your step. Let's get to some coding!

The `partitionMapper` only has to implement one method, `mapPartitions`. That method needs to return an instance of an object that implements the `PartitionPlan` interface. Conveniently, JSR-352 implementations provide you with one called, not surprisingly, `PartitionPlanImpl`. Just new up an instance of that and you're on the way.

We'll presume you've figured out how much data you have and how many partitions you want and how many you want to let run at once. If not, go back and re-read last week's post and ponder some more.

Once you're set, you can call the `setPartitions` and `setThreads` methods to set those values into the plan you instantiated.

In the JSL for the partition you'll need to have set up the properties that will get injected into each copy of the step. If you have a chunk step, the reader might need to know what record key values to start and stop at. For our sample, the batchlet needs to know the full path of the file it should process. Decide what the property names are and put them in the JSL.

Now you need to set up the different property values to give to each instance of the step. Start by creating an array of Properties objects. You'll need a slot in the array for each partition. Remember the array size is the number of partitions, but partition numbers start at zero (like the array index will).

At this point, iterate through the array, creating a new Properties object to go in each array slot. Then use the `setProperty` method to set whatever properties you need for each partition. You might be tempted to use the `put` method, but don't do it. Properties have to be Strings. Using `put` will allow you to set integers or other things in there that won't work properly, and you'll get a null value injected into your partitions. The `setProperty` method won't let you set anything but Strings.

We have a couple more things to talk about, but we know enough to go have a look at the code. The JSL is in UsingPartitionMapper.xml and the mapper itself is SamplePartitionMapper.java. Our step is a batchlet that pretends to do some processing with the file injected into it and is called FakeFileProcessor.java for that reason.

The sample parts are here: https://github.com/follisd/batch-samples

Java Batch How-To: Use a PartitionMapper – Part 3

If you've taken a look at our sample `PartitionMapper` from last week you might have noticed a couple of things we didn't talk about relating to values set into the `PartitionPlan`.

The first of these is a couple of places where we set the partition count to zero using the `setPartitions` method. This is a special case you might need to consider. What should you do if you get into your mapper and find it has no data to process? If this is a surprising condition (what do you mean the table of store locations is empty?) then you probably want to throw an exception and fail the job. Something is obviously horribly wrong.

But there might be cases where this is totally normal. Suppose this job runs every day and processes any files it finds in a certain location. It might be completely reasonable for there to be no files there (maybe today is a holiday). In that case, you really didn't want to run this step at all. You could have a batchlet step that runs before the partitioned step to figure out if there is any data to process and use a step exit status value and flow control to jump over the partitioned step.

Or you can just let it go and get into the mapper to discover there is no data. Set the partition count to zero in the plan and return it. You don't need to set any other data into the plan. The batch container will see the zero partition count value and just end the step. Any defined step listeners will run, of course, so you should be sure they can handle the step not having actually done anything.

The second thing you might notice is our call to the `setPartitionsOverride` method. This has to do with a previously executed job being restarted. If restart processing involves running a partitioned step that has executed already (maybe only partially) in a previous execution, this method tells the container what to do about partitioning.

If you set the partition override value to true, then that tells the container to ignore anything that happened before and just use the partition values found in the partition map you are creating. You want to do this if you just need to start completely over, or if the data you are processing will have been modified so you won't process it again.

Suppose a partition's job is to process a particular file. When it is done with the file, it deletes it (or moves it somewhere else). In that case you would just work with the files you find to process because only the ones that are left need processing (yes, there are some windows here you'd need to worry about).

Do you want to say true or false here? It really depends on how your application is going to handle a restart. Are you going to re-process data you already processed? Can you tell? Will checkpoint data in the partitions help you? If so, you want to say false and use the existing partitioning information so checkpoint data will be used. It is a tough call to make and you need to understand how your application will handle a re-start to set it to the right value. Be careful because it can matter a lot.

Java Batch How-To: Write a Retryable ItemReader

How you do this depends a lot on how your reader functions, so this time we're going to tie the discussion a little more closely to the actual example. As usual, the code is all at https://github.com/follisd/batch-samples

Our JSL is defined in CodingARetryableReader.xml, but you don't need to go look at that yet. First let's talk about how our reader is going to work. We're going to presume that we need to read records out of some database and that we're using an integer as the primary key. We'll start with a value of one and increment our way up until we get to the last valid key (hard-coded as 100 in our example just to keep things simple).

That means when our reader is called to do open processing we'll initialize the integer index to one and as we read items we'll increment the index. Sounds easy and simple. Now we're going to cause trouble. The processor for our sample, ThrowOnThirtyEightProcessor will throw an Illegal Argument Exception when it is passed an object from the reader for index 38. It will only do this once, so the error will be 'resolved' if we retry. We'll add a retryable exception element to the JSL to let the batch container know we expect this exception and want to just retry it.

What happens next? The batch container will roll back the transaction wrapping the chunk processing, close the reader and writer, and then re-open them. The expectation is that the reader will start over at the last checkpoint. What's that? Our reader didn't provide any checkpoint data? Oh dear. That means when the reader is called to do open processing it will initialize the index at one just like when the job started. We'll re-process the records we've already processed and committed. We need to provide some checkpoint data.

To do that we need to create a serializable object to contain the reader's checkpoint data. You might be tempted (I was) to make an inner class within the reader to represent the data. After all, nobody else cares about this except the reader. However, if you do that you'll get in trouble when the batch container tries to serialize or deserialize the data. Make it a separate class. Ours is called IntegerReaderState because it just contains an Integer.

After we update our reader to provide the state object as checkpoint data, there's just one thing left. We need to update our open processing to determine whether we've been passed checkpoint data or not. When the job first starts, the parameter is null, but after a retry or restart the parameter will contain the last checkpoint data provided. If the parameter is null we need to create the checkpoint data object, but if one was passed in then that's the data to use so we pick up where we left off.

If you run the sample job, you should also notice that after the exception at index 38 and the retry back to the last checkpoint (at 30 because we checkpoint every 10 elements), the batch container will take a checkpoint after every record until it gets past the one that caused the problem. So checkpoints will happen for element 31, 32, 33, etc. up to 38. Then the container will resume checkpointing every 10 elements as before.

Java Batch How-To: Use a Collector/Analyzer

The Collector/Analyzer pair can be used to communicate between the concurrent threads of a partition and the main step thread. The usual example is that the partitions are counting something, and purpose of the step is to get a total count. Each partition needs to report its results to the main step thread which will add up the results as the partitions complete.

The Collector runs on the partition thread, but you need to figure out a way for the partition processing to communicate with it. An easy way is to use the `StepContext's` Transient User Data. If the partitioned step is a batchlet, the Collector will get control when the batchlet finishes. If the partitioned step is chunking then the Collector will get control after each chunk. Note that the Collector gets control after the transaction commits (as opposed to the Chunk Listener's `afterChunk` method which gets control after the Writer but before the transaction commits).

For our example we'll just use a batchlet. We've re-used the `SamplePartitionMapper` from earlier to provide a list of .txt files found in an input directory as a property to each partition. As before, our batchlet (`UserDataPartitionedBatchlet`) just logs the file name it received, but it also sets that file name into a String set into the Transient User Data for the step. The batchlet also sets an Exit Status for the partition that includes the thread identifier (just to have something probably different in there from partition to partition – although it could repeat if you have a lot of partitions).

Having information in the Transient User Data allows our Collector (`UserDataCollector`) to get to it and set it as the returned value from the `collectPartitionData` method. This has to be a Serializable value so you may need to create a separate Java class to contain the results. In our case it is just a String so nothing special is needed.

Our Analyzer (`UserDataAnalyzer`) gets the Batch Status and Exit Status value from each partition in the `analyzeStatus` method. Our Exit Status values contain the thread id of the thread used to run each partition so there should at least be some variety in the values. The `analyzeCollectorData` method will see all the data returned by the Collectors running for each partition. In our example it will contain the file name processed by each partition.

An important thing to remember about all this processing is that it is not transactional or hardened in any way. If the job fails or some other bad thing happens after a chunk or a batchlet completes there is no guarantee the Collector will run or that data returned by a Collector will ever be seen by the Analyzer. Any in-flight information is lost and a restart of the job won't recover it.
As usual, the code is all at https://github.com/follisd/batch-samples.

Kubernetes Jobs - Introduction

Let's take a break from Java Batch and look at something related but totally different, just for a little bit. I was reading about Kubernetes Jobs and trying to figure out how it might interact with Java Batch. I thought I'd share what I learned. To be clear, I'm not an expert in Kubernetes or Kubernetes Jobs, so there's a fair chance I got something wrong or at the very least that I'm oversimplifying something. And, of course, I'm just rambling along here and not endorsing it or anything like that.

So Kubernetes Jobs is still just some YAML that you use with kubectl to get pods created and running your stuff. In the YAML you specify a 'kind' value of 'Job' and there's a spec section where you put all your stuff telling it what you want done. Down in there is the 'command' where you actually specify the thing that's going to happen when all this gets spun up.

And what might that be? Well, it is probably a script that's going to run a bunch of stuff that includes whatever your job is. Maybe it starts a JVM to run some Java stuff or maybe it runs something else entirely. If you run some Java, does that Java implement the whole 'job'? Maybe. The Kubernetes Jobs stuff has a few different models you can use depending on how you think about what you want each instance to do. We'll get into that in the coming weeks.

A very important thing about Kubernetes Jobs seems to be how the command completes. The returned value from the command indicates whether the instance was successful or not. That's going to tie into restart handling and a bunch of other stuff.

Well, what would that return value be? If you ran a script it is going to be the return value from the script. Which could be whatever you want. If you launched a JVM then the JVM would have exited with a completion code and maybe you could propagate that back out, if that value indicates whether your application was successful or not.

Of course, if you started a Liberty server to run a JSR-352 batch job inside the pod, the server will just stay up after the job completes. You'd need to find some way to indicate to the controlling script that the job was finished and it should take down the server. And some way to propagate the exit status value (a String remember) into some sort of returned value from the script that launched it all.

Well, all we've done so far is use Kubernetes Jobs to run something and we've got some stuff to think about. Next week we'll look at some of the simpler controls.

Kubernetes Jobs – Restart, Backoff, and Deadline

As a reminder, we're taking a casual stroll through Kubernetes Jobs. This week we'll look at three configuration values you can specify that influence how it behaves.

The first one is the restart policy. We said last week that the Jobs processing depends a lot on the return value from the command you execute. As part of the YAML you can also specify a restartPolicy value of Never or OnFailure.

Obviously Never will never restart and OnFailure will restart on a failure. But what is a failure? Well, whenever your command processing returns something bad. Or if some other failure occurs (like the Pod crashes or is stopped through some other Kubernetes processing). If your application just blew up and so the whole thing failed, then OnFailure will spin it back up again and you can pick up where you left off. But if the whole thing died you might not have any little footprints you left about where you were in processing (depending where you kept those). As we've seen in earlier posts, restart processing is tricky. Just having Kubernetes Jobs restart your command is only the beginning.

Backoff is another interesting configuration option. This ties into the restart processing. With Backoff you can configure the number of times it will retry. That's a good thing so you don't end up endlessly retrying something that won't ever work. This is similar to the retry limits in the JSR-352 specification. Although in this case we're potentially restarting your whole environment. There was an interesting note in the documentation about Backoff processing also adding in increasing delays between restarts on every try. I guess the idea is that maybe a failure left some stuff that needs cleaning up or maybe you're hoping some problem will get resolved and it waits longer and longer between retries hoping things will sort themselves out (maybe it needs some other automation to restart something else).

Finally, I wanted to look at the activeDeadlineSeconds configuration value. This value applies to the whole job, no matter how many Pods or containers got started or restarted. This is basically a timeout value to get the job done. When you exceed this value all the Pods are terminated and the job fails. Bang…over. No gentle quiescing or anything. So while it seems like a good idea to have this drop-dead time value set to keep things from just running away, it sounds like it is a pretty violent end if you go over it. So set it high.

Alright, so all that's kind of interesting, but next week we'll start into the good stuff and look at our first model for running a job.

Kubernetes Jobs – One Pod, One Job

There are two values that control concurrency and completion for a job. These are spec.completions and spec.parallelism. For our discussion this week we're going to let them both default to one.

That means you're going to get one pod started to run the command you specified. If it finishes successfully, then you're done. If it doesn't, then (depending on the restart setting we talked about last week) it might get restarted until it works.

This model is a very traditional way of looking at things and so it makes sense to default to this. In the following weeks we'll look at some interesting variations where we change these two configuration values.

So this means that whatever command you issue, it has to startup something that is going to do the entire work for the job, even if it consists of multiple 'steps' (whether those are JSR-352 steps or just different things the script your run will do). Success or failure is based entirely on the returned value which has to cover the whole thing.

This is just like the exit status from a JSR-352/Jakarta Batch job.

However, if you're going to say the job failed, it might get restarted. Of course, restarting a JSR-352 job includes remembering how steps that have run completed and checkpoint data if a chunk step was in-flight when it failed. If you've just got a script doing a bunch of stuff, you're going to have to leave yourself some notes about what worked and what you want to retry.

Or you can just fail and leave it to somebody (or some automation) to figure out. Maybe some higher up thing runs some different job if this one fails to try and clean things up?

I'm pretty traditional so I'm usually in favor of large multi-step jobs that do a bunch of things, implementing an entire process perhaps. But done this way where the application is responsible for so much of error handling it seems like maybe it would be better to break it up a little and let something external handle some of the things that seem more infrastructure than application.

Or just spin up a Liberty server and let it to do it, but you need to be sure things like checkpoint data in the Job Repository survive the failure and restart of the pod running the server that's running the job.

Kubernetes Jobs – One Pod, One Record?

Last week we talked about letting Kubernetes Jobs default to just one pod with one success. This time we'll look at making use of the spec.completions configuration to deliberately run the 'job' more than once.

Ok, so we aren't really running the job more than once. We're running the command that is specified in the YAML more than once. We use the completions configuration to tell Kubernetes how many successful completions we need in order for the job to finish.

Say what? I have to run the job multiple times successfully for the job to succeed? No, just the command. The set of successful commands is the job. Think of each execution of the command as one pass through a loop. Think of this Kubernetes Job as a single chunk step job from Jakarta Batch.

The pod will get spun up and run the command you specify which does one instance of the thing you want done. Maybe it reads a single record from an input file and process it. If it works, the command completes successfully. Then we go around again. Until we've processed spec.completions number of records successfully.

That's a bit odd too. You'd need to know how many records you had to process up front. And if you specify that number as the number of successful completions then you must successfully process them all. A failure won't count so you'd have to try again with that record to finish.

And it isn't an "at least" count. If you've got 1000 records and figure 950 successes is enough, it will get to 950 and stop whether there are still 50 records to go or not. Of course, if you're just working off some backlog that would be ok. Every time this Job is run, we process N records. As long as there are least N records to process (probably more if you allow for failures) then it would probably be ok.

I think this feels a little quirky because I'm trying to force this into what I think of as a 'job'. Maybe you wouldn't use this to process records. Maybe it just does something and after it has done that successfully 5 times (or whatever) that's good enough. Maybe your job has a list of 100 people it can notify about something, and it needs to successfully notify 10 of them. Each execution of the command to do the notification gets a name from a list and gives it a try. Once you've successfully notified 10 people you're done. You don't care which ten you got. Something like that perhaps.

Regardless, it is an interesting model.

Kubernetes Jobs – Indexed Completion Mode

As we continue our look at Kubernetes Jobs, this week we explore yet another configuration option: spec.completionMode. You can specify one of two values: Indexed or NonIndexed. The default (and the behavior we've been assuming so far) is NonIndexed.

Ok, so what happens if you specify an Indexed Completion Mode? Well, it ties into the value you specify for spec.Completions. Each started pod is assigned an index value between zero and spec.Completions minus one (so if you set Completions to 5 you'll get index values from zero to four). To finish the job you need one successful completion for each index value.

This is really kind of the same as just setting Completions to a value bigger than one. You need that many successful completions to finish. The difference is just that each started pod gets an input value that tells it which completion it is trying to achieve. And you could use that value as an index into something.

When we talked about Completions bigger than one last week, we said that you might process through a bunch of records trying to successfully process some number of them. With indexing you could use the index to tell you which record you are trying to process, not just whatever is 'next' (however you are keeping track of that on your own).

Your pod would get started with an index of zero to process the zeroth (is that a word? Apparently.) record. It would, if configured, get restarted over and over until it completed successfully when another pod would get started to process the first record, and so on.

Or, the index might just be a key that each started instance uses to find parameters that tell it what range of records to process. I guess in our discussion last week that might have been an option too…each started pod might process 1000 records (instead of just the one I discussed). Using indexes as a key into some sort of parameterization table might let you customize the range. But you do still need to know in the YAML how many successful completions you need.

In that sense it feels a lot like having JSR-352 JSL for a partitioned step with the partition plan in the JSL. You have to know from the start what you're going to do. Well, and a partitioned step runs things concurrently. What we've described here all happens one at a time. Next week we'll get concurrent…

Kubernetes Jobs – Concurrency

Finally, we're going to talk about concurrency with Kubernetes Jobs. This is done by just setting spec.parallelism to the maximum number of concurrent pods you want to allow. The default is obviously one. As with concurrency in Jakarta Batch, this doesn't mean that you will always get this many concurrent pods running at once, just that you won't get any more than this. Usually.

In a case where you want more than one successful completion (indexed or not) you could specify parallelism up to the number of completions and get multiple copies of the pod running at the same time (at least possibly) until you reach the required number of successful completions. This is much more analogous to the partitioned step from Jakarta Batch.

But the documentation for Kubernetes Jobs goes one step further and encourages you to think of each pod instance essentially as a server. The pods get started concurrently and the completion value is equal to whatever parallelism value you set. Each pod spins up and starts processing whatever it processes, somehow coordinating with the other instances.

The tricky part is shutting down. Somehow the different pods need to figure out that it is time to go away. Maybe when they run out of data? Anyway, as soon as a pod ends successfully, that counts as a completion, so no new pods get created. Presumably the other pods are ending too. Once all of them are done you've reached your completion count and the job is over.

Of course, if something bad happens and a pod completes unsuccessfully, a new one is started to run that part of the job again.

This seems like kind of a big deal to the Kubernetes Job stuff, but it feels a whole lot like just a set of servers based on some cardinality value and Kubernetes is keeping that many servers active.

I mean sure, you could say the work being done is part of a 'job' because they are reading records or processing database rows in some range and not handling HTTP requests. And maybe, because the 'servers' end when they figure out they've run out of things to do (something HTTP servers don't usually do), it is kind of a job.

But it feels like a bit of a stretch to me.

Well, hey, if that model fits your needs, call it whatever you want.

This wraps up my pondering of Kubernetes Batch. Lots of doc is available online if you're interested in following up.

Java Batch – Batch Jobs in Java?

When I started this blog over three years ago I began with this question. In the coming weeks I'll be looking back over all the things we've covered and reconsider the question. I'll take it in parts and this week we'll just look at Java itself.

So, whether you're writing some new application and trying to decide what language to use, or you've got an existing application you're considering rewriting, should you consider Java? Rewriting existing applications is expensive and introduces risk. You need to be sure you understand why you're doing it. Don't just wave your hands and say 'modernization'.

But either way, our question here is whether you should consider Java for batch applications. I certainly think so. I'm not saying that Java is the only language to consider, but it is certainly a good option.

Batch applications need access to data sources. They read it from somewhere and write things somewhere. Java has mechanisms to access pretty much anything, including some fairly obscure things (well, ok, my obscure might be your everyday data and vice versa, but still).

Being object oriented makes Java pretty good at handling a 'row' of data as attributes of an object. It just fits pretty well into the language.

Performance is always a concern with batch. It can't take days to run your hourly job. Here again Java is pretty good because the JIT does a better and better job of optimizing the generated object code the more it runs the same code, and batch applications do tend to run the same code over and over again.

And, of course, Java is supported pretty much anywhere you might want to run your batch application, so you aren't tied to a particular platform. That lets you make a platform decision based on other considerations (reliability, cost, performance, etc.). I'm not going to get into that argument here. My point is that choosing Java as a language for batch removes language choice as a factor in choosing a platform…although you might consider that Java runs better some places than others (looking at you IBM Z).

Ok, so I think we've shown over the last three years that the answer to this first question is probably 'Yes'. Next week we'll dig in a little farther..

Java Batch – Should You Use a Framework?

So you've decided that you might want to write (or rewrite) some batch applications in Java. The next question is whether you just want to sit down and write it or if you want to make use of a framework. There are several and, as you might imagine, I'm a bit biased toward JSR-352/Jakarta Batch. But which framework isn't our question..for this week we're just looking at whether you should consider any framework or just write it yourself.

I'd like to start by making a distinction between using a formal framework and just stealing/copying code. Even before the days of the internet and readily available code samples, people stole code (or called it reuse). Probably everybody has, at one time or another, asked around to see if anybody had some code that did <this thing you needed to do>. Maybe because you didn't know how or maybe you were just hoping to avoid having to type it all in. This is all just fine (well, up to a legal point anyway), but not what I'm talking about with a framework.

A framework, to me, is a formal pile of code intended to make whatever you're doing easier, every time you have to do it. Even if you only intend to do something once, having a framework can make it easier.

Of course, on the other hand, having to figure out how to use a framework and sometimes force what you need to do into the structure the framework requires can end up being extra work. Especially if you think you're only going to do this once.

So what to do? Well, sure if you're just throwing together some quick little thing that is mostly code lifted from other things, the risk is low, you get results fast and all is good. For now anyway. Over time those quick little things tend to grow, and get more complicated. Somehow that temporary utility you wrote to hold things together until something fancier gets done ends up still being used, maybe for something else entirely, 10 years later.

And thus I tend to come down on the framework side. It might be a pain to get started with, but once you're past those initial hurdles the next thing will go easier. And there's a bunch of code you don't have to write and, more importantly, not maintain.

Frameworks are not only maintained by somebody else, they are (usually) used by a lot of different people for different sorts of things and that tends to shake out bugs, making it more stable for everybody. And perhaps the risk of breaking a lot of people tends to make those writing fixes (or new features) a bit more cautious.

You could stretch this argument and say it means you should only buy software off the shelf and never write your own stuff. I'm not saying that. At some point your particular needs take over, but there's a lot of stuff there's no reason to write again and again. Let the framework do the drudgery and own the stuff that's unique to your situation.

Java Batch – An Open or Proprietary Framework?

Last week we, ok, I, concluded that in a lot of cases using a framework of some sort to help write Java Batch applications is probably a good idea (enough weasel words in there for everybody?). In essence, a framework gives you reusable code to do a lot of the things you'd have to do yourself, and maintenance of that code is separate from the stuff you have to write.

This week we'll go one step further and consider whether you should prefer an open or proprietary framework. Obviously I'm thinking of JSR-352/Jakarta Batch as the open framework. There are a few different proprietary ones (you know who you are) and my intention here isn't to get into a debate about the differences between them.

This is basically an open vs. proprietary question. And we're only really considering this if you're starting fresh. If your environment has a long history with one of the proprietary frameworks, you're unlikely to just suddenly switch everything over.

Or maybe one of the proprietary frameworks has some feature that you just have to have and it isn't in Jakarta Batch. Well, now that's an interesting point. If you're using a proprietary framework and there's some feature you need that it doesn't have, all you can really do is open a requirement against whoever owns it, maybe enlist some other users of that framework to join in, and try to put some pressure on the owner to do the work.

An open framework is open to enhancements. You may need to do the same political work to get others interested, and maybe somebody else will be interested enough to even do the work. But you can also do the work yourself. That's not really an option with most proprietary frameworks.

I'm not going to rehash all the arguments in defense of open source etc. But I will point out that even if you're using an open source framework, you still need to get support from somewhere. If it is always going to be you trying to figure out what the problem is when bad things happen, maybe you are better off writing it all yourself. For production mission-critical applications, you really want somebody you're paying to be on the hook to help get things working again.

There are clear benefits to open source, but being able to get support when you need it is also really important.

Java Batch – Is Liberty a Good Place to run Jakarta Batch?

If you've decided this is the programming model you want to use, you've still got several choices about where to run it. To start with there are several different implementations of the specification. The original reference implementation is the basis for some of them (including the one in Liberty). In some cases you can run Jakarta Batch in a simple stand-alone JVM, while in others you're part of a larger server environment (which would be the case with Liberty).

What you choose is probably going to depend a lot on what you're already doing. If, for example, you've already got a lot of Liberty servers running OLTP workloads, it would probably make sense to run the batch workloads in Liberty.

Another factor might be how you intend to manage the JVMs that are running the jobs. Are you going to start a JVM to run a job, then tear it down when the job is done, or keep it around as a job-server (for z/OS folks, think of it as a JES initiator).

On the one hand, starting and stopping a lot of JVMs seems like unnecessary overhead, but that's more in keeping with a containerized environment where you might provision an instance of an image to run a job. But, of course, you could provision an instance of a job-server that will run lots of jobs taken from a queue or something like that. Our discussion of Kubernetes Jobs made clear you've got a lot of options.

Hopefully, if you're reading this thinking I'll tell you what the 'best' approach is, you've figured out that I'm not going to do that. As with most things, there are a lot of different approaches you can take and you need to weigh your own particular needs against the pros and cons of each one. And what is a 'pro' to me might not be to you (and vice versa).

But our question isn't actually what the best environment is any more than it is what the best platform or best language is. The question is whether Liberty makes a good option. Obviously I think it does or I wouldn't have spent months of blog posts on our implementation of the specification. I think having the infrastructure of the server, with all its other features, including integration with z/OS, around the batch container make it a very viable option for your Java Batch workloads.

When I started writing this I thought I was going to get into a long list of the benefits of Liberty, but looking back over it I can see all I've done is wave my hands around a lot and told you to reach your own decision based on your own needs. All in all that's probably the best advice I can give…

Java Batch – Wrapping Up

Well, when I made the first post in this series way back in August of 2018 I had no idea how long this would go on. I had sort of a vague idea of talking about the details of the spec for a few weeks, then focus a bit on the Liberty implementation, maybe a bit on the z/OS specific stuff, and then wrap it up. If I was lucky I'd stretch it to maybe early in 2019. I wondered if maybe I should post monthly instead of weekly to spread it out a bit more.

Every time I felt like I was running out of things to talk about, something new would surface and feed another whole series of posts. But I think we're at an end now. Plus, to be honest, I'm getting a bit tired of writing about this.

That said, there's actually some cool stuff going on with the Jakarta Batch specification. In a series of earlier posts we looked at some potential updates and enhancements, but that was basically just me picking ones I thought would make an interesting post.

If you're interested in finding out what's going on, or just want to poke around a bit, you can check things out at the two links below. There's also a mailing list you can subscribe to (and a Slack channel, although that's not very active).

https://projects.eclipse.org/projects/ee4j.batch

https://jakarta.ee/specifications/batch/

Finally, I am rolling up the text of all the posts into a single PDF that you can find at the link just below. It also includes the source from the samples I linked to in github (on the off chance the PDF outlives it). Also, while advertising these posts in LinkedIn, I started having an associated song for each post (the song for this post is "The End" by The Doors). The PDF contains the complete list of songs.

It has been fun…thanks for reading….

https://www.ibm.com/support/pages/node/6560420

Appendix – HowTo Source Code

A series of posts provided 'how to' advice and commentary and included actual source samples located in github here: here: https://github.com/follisd/batch-samples

For sake of completeness, all the same source is included in the following pages.

BatchletSetExitStatusOk

```
package batch.samples;

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

import javax.batch.api.Batchlet;
import javax.batch.runtime.context.StepContext;
import javax.enterprise.context.Dependent;
import javax.inject.Inject;

@Dependent
public class BatchletSetExitStatusOk implements Batchlet {

     private static final Logger log = Logger.getLogger( Batch-
letSetExitStatusOk.class.getName() );

     @Inject
     private StepContext stepContext;


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

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

     /**
     * @see Batchlet#process()
     */
    public String process() {
          log.log(Level.INFO, "Batchlet in step "+stepContext.getStep-
Name()+" setting exit status to OK");
          stepContext.setExitStatus("OK");
               return null;
    }

}
```

BatchletSetJobStatus

```
package batch.samples;

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

import javax.batch.api.Batchlet;
import javax.batch.runtime.context.JobContext;
import javax.enterprise.context.Dependent;
import javax.inject.Inject;

@Dependent
public class BatchletSetJobStatus implements Batchlet {

     private static final Logger log = Logger.getLogger( BatchletSet-
JobStatus.class.getName() );

     @Inject
     private JobContext jobContext;


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

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

     /**
     * @see Batchlet#process()
     */
    public String process() {
     String oldStatus = jobContext.getExitStatus();
     String status = new String("Batchlet Set Job Exit Status");
     jobContext.setExitStatus(status);
          log.log(Level.INFO, "Batchlet setting job exit status from
*"+oldStatus+"* to *"+status+"*");
               return null;
    }


}
```

BatchletTransientReceiver

```
package batch.samples;

import java.util.logging.Level;
import java.util.logging.Logger;
import javax.batch.api.Batchlet;
import javax.batch.runtime.context.JobContext;
import javax.batch.runtime.context.StepContext;
import javax.enterprise.context.Dependent;
import javax.inject.Inject;

@Dependent
public class BatchletTransientReceiver implements Batchlet {

	private static final Logger log = Logger.getLogger( BatchletTran-
sientReceiver.class.getName() );


	@Inject
	private JobContext jobContext;
	@Inject
	private StepContext stepContext;

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

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

	/**
     * @see Batchlet#process()
     */
    public String process() {
	String data = (String)jobContext.getTransientUserData();
		log.log(Level.INFO, "Batchlet in step "+stepContext.getStep-
Name()+" received data *"+data+"*");
			return null;
    }

}
```

BatchletTransientSender

```
package batch.samples;

import javax.batch.api.Batchlet;
import javax.batch.runtime.context.JobContext;
import javax.enterprise.context.Dependent;
import javax.inject.Inject;

@Dependent
public class BatchletTransientSender implements Batchlet {


     @Inject
     private JobContext jobContext;


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

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

     /**
     * @see Batchlet#process()
     */
    public String process() {
     jobContext.setTransientUserData(new String("Hello from Step1"));
               return null;
    }

}
```

JobListenerSetJobStatus

```
package batch.samples;

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

import javax.batch.api.listener.JobListener;
import javax.batch.runtime.context.JobContext;
import javax.enterprise.context.Dependent;
import javax.inject.Inject;

@Dependent
public class JobListenerSetJobStatus implements JobListener {

     private static final Logger log = Logger.getLogger( JobListen-
erSetJobStatus.class.getName() );

     @Inject
     private JobContext jobContext;


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

     /**
     * @see JobListener#afterJob()
     */
    public void afterJob() {
     String oldStatus = jobContext.getExitStatus();
     String status = new String("Job Listener Set Job Exit Status");
     jobContext.setExitStatus(status);
          log.log(Level.INFO, "Job Listener setting job exit status
from *"+oldStatus+"* to *"+status+"*");
    }

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

}
```

RandomIntegerProcessor

```
package batch.samples;

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

import javax.batch.api.chunk.ItemProcessor;
import javax.enterprise.context.Dependent;

@Dependent
public class RandomIntegerProcessor implements ItemProcessor {

     private static final Logger log = Logger.getLogger( RandomInte-
gerProcessor.class.getName() );

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

     /**
     * @throws SampleSkippableException
       * @see ItemProcessor#processItem(Object)
     */
    public Object processItem(Object arg0) throws SampleSkippableEx-
ception {

     Integer value = (Integer)arg0;
          if (value.intValue()>100) {
               // Reader generates random values 0-110
               // For our purposes we'll consider anything over 100
to be bad and skip it

               log.log(Level.INFO, "Processor skipping value-
>"+value);

               SampleSkippableException sse = new SampleSkippableEx-
ception();
               sse.badValue(value);
               throw sse;
          }
          return arg0;
    }

}
```

RandomIntegerReader

```
package batch.samples;

import java.io.Serializable;
import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.batch.api.chunk.ItemReader;
import javax.enterprise.context.Dependent;

@Dependent
public class RandomIntegerReader implements ItemReader {

     private static final Logger log = Logger.getLogger( RandomInte-
gerReader.class.getName() );

     private Random rand = new Random();
     private int itemsRead = 0;
     private int maxItemsRead = 100;

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

     /**
     * @see ItemReader#readItem()
     */
    public Object readItem() {

     int value = rand.nextInt(110); // Random between 0-110

     Object retVal = new Integer(value);

     // Something to stop the loop
     ++itemsRead;
     if (itemsRead>=maxItemsRead) {
          retVal = null;
     }

          log.log(Level.INFO, "Reader returning->"+retVal);

     return retVal;

    }

     /**
     * @see ItemReader#open(Serializable)
     */
    public void open(Serializable arg0) {
    }

      /**
     * @see ItemReader#close()
     */
    public void close() {
    }

      /**
     * @see ItemReader#checkpointInfo()
     */
    public Serializable checkpointInfo() {
                return null;
    }

}
```

RandomIntegerWriter

```
package batch.samples;

import java.io.Serializable;
import java.io.Writer;
import java.util.Iterator;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.batch.api.chunk.ItemWriter;
import javax.enterprise.context.Dependent;

@Dependent
public class RandomIntegerWriter implements ItemWriter {

     private static final Logger log = Logger.getLogger( RandomInte-
gerWriter.class.getName() );

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

     /**
     * @see ItemWriter#open(Serializable)
     */
    public void open(Serializable arg0) {
    }

     /**
     * @see ItemWriter#close()
     */
    public void close() {
    }

     /**
     * @see ItemWriter#writeItems(List<java.lang.Object>)
     */
    public void writeItems(List<java.lang.Object> arg0) {
     Iterator<java.lang.Object> iterator = arg0.iterator();

     while (iterator.hasNext()) {
          Integer value = (Integer)iterator.next();
          log.log(Level.INFO, "Writer received->"+value);
     }
     log.log(Level.INFO, "--------------------");
    }
```

```
      /**
      * @see ItemWriter#checkpointInfo()
      */
    public Serializable checkpointInfo() {
                return null;
    }

}
```

RetryableIntegerSequenceReader

```
package batch.samples;

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

import javax.batch.api.chunk.ItemReader;
import javax.enterprise.context.Dependent;

import batch.samples.helpers.IntegerReaderState;

@Dependent
public class RetryableIntegerSequenceReader implements ItemReader {

	private static final int maxInt = 100;

	private IntegerReaderState rs;

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

	/**
	 * @see ItemReader#readItem()
	 */
    public Object readItem() {
	Integer retVal;

	// If we're not done, get the current 'record'
	if (rs.curInt().intValue()<=maxInt) {
		retVal = rs.curInt();
		// position for next read
		rs.curInt(new Integer(rs.curInt().intValue()+1));
	} else {
		retVal = null;
	}


	return retVal;

    }

	/**
	 * @see ItemReader#open(Serializable)
	 */
    public void open(Serializable arg0) {
      if (arg0==null) {
           rs = new IntegerReaderState();
      } else {
           rs = (IntegerReaderState)arg0;
      }

    }

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

      /**
     * @see ItemReader#checkpointInfo()
     */
    public Serializable checkpointInfo() {
                return rs;
    }


}
```

SampleCheckpointAlgorithm

```
package batch.samples;

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

import javax.batch.api.chunk.CheckpointAlgorithm;
import javax.enterprise.context.Dependent;

@Dependent
public class SampleCheckpointAlgorithm implements CheckpointAlgorithm
{

      private static final Logger log = Logger.getLogger( SampleCheck-
pointAlgorithm.class.getName() );


      private long chunkStartTime;
      private long startTimeThisIteration;
      private long totalTimeThisChunk;
      private int iterationCount=0;
      private int chunkCount = 0;
      private long totalWriteTime;

      private long chunkTimeGoal = 5000; // 5 seconds

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

      /**
      * @see CheckpointAlgorithm#checkpointTimeout()
      */
     public int checkpointTimeout() {
      return 30;  // just a backstop in case things go wrong
     }

      /**
      * @see CheckpointAlgorithm#endCheckpoint()
      */
     public void endCheckpoint() {
      long currentTime = System.currentTimeMillis();
      long writeTime = currentTime - chunkStartTime - totalTime-
ThisChunk;
      ++chunkCount;
      totalWriteTime +=writeTime;
 //        log.log(Level.INFO, "ChunkEnded - writeTime = "+writeTime);
```

```
      log.log(Level.INFO, "ChunkEnded - iterations = "+iterationCount+"
missed target by = "+(currentTime-chunkStartTime-chunkTimeGoal));
     }

      /**
      * @see CheckpointAlgorithm#beginCheckpoint()
      */
     public void beginCheckpoint() {
      totalTimeThisChunk = 0;
      iterationCount = 0;
      chunkStartTime = System.currentTimeMillis();
      startTimeThisIteration = chunkStartTime; // initialize for first
pass
//         log.log(Level.INFO, "Chunk started at "+startTimeThisItera-
tion);

     }

      /**
      * @see CheckpointAlgorithm#isReadyToCheckpoint()
      */
     public boolean isReadyToCheckpoint() {
      boolean ready;

      ++iterationCount;

      long currentTime = System.currentTimeMillis();
      long timeThisIteration = currentTime - startTimeThisIteration; //
how long for this pass?
      startTimeThisIteration = currentTime;  // reset for next itera-
tion
      totalTimeThisChunk += timeThisIteration; // update running total

//         log.log(Level.INFO, "isReadyToCheckpoint - timeThisIteration
= "+timeThisIteration);
//         log.log(Level.INFO, "isReadyToCheckpoint - totalTime-
ThisChunk = "+totalTimeThisChunk);

      long writeGuess;

      if (chunkCount==0) { // first time, no idea how long the write
will take
           writeGuess = 300; // guess...
      } else {
           writeGuess = totalWriteTime/chunkCount; // average write
time
      }

//         log.log(Level.INFO, "isReadyToCheckpoint - writeGuess =
"+writeGuess);
```

```
     long estimatedChunkTime = totalTimeThisChunk + writeGuess;

//         log.log(Level.INFO, "isReadyToCheckpoint - estimatedChunk-
Time = "+estimatedChunkTime);

     if (estimatedChunkTime > chunkTimeGoal) {
          ready = true;
     } else {
          ready = false;
     }

//         log.log(Level.INFO, "isReadyToCheckpoint - ready = "+ready);


     return ready;
    }

}
```

SampleDecider

```
package batch.samples;

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

import javax.batch.api.Decider;
import javax.batch.runtime.StepExecution;
import javax.batch.runtime.context.StepContext;
import javax.enterprise.context.Dependent;
import javax.inject.Inject;

@Dependent
public class SampleDecider implements Decider {

     private static final Logger log = Logger.getLogger( SampleDe-
cider.class.getName() );


     // All these have to match the values in the JSL
     private static final String okInputStatus = new String("OK");
     private static final String badInputStatus = new String("BAD");

     private static final String okOutputStatus = new String("OK");
     private static final String badOutputStatus = new String("BAD");
     private static final String mixedOutputStatus = new
String("MIXED");
     private static final String unknownOutputStatus = new String("UN-
KNOWN");

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

     /**
     * @see Decider#decide(StepExecution[])
     */
    public String decide(StepExecution[] arg0) {

     // The rules are:
     // - If all input status values are ok, then output is ok
     // - If all input status values are bad, then output is bad
     // - If a combination, then output is mixed
     // - If we don't find either one, output is unknown, stop the job

     String outputStatus = unknownOutputStatus;
     boolean foundOk = false;
```

```
      boolean foundBad = false;

      // Loop through the input status values
      for (int i=0; i<arg0.length;i++) {
            String flowStatus = arg0[i].getExitStatus();

            log.log(Level.INFO, "Decider evaluating exit status "+i+"
from step "+arg0[i].getStepName() +" value="+flowStatus);

            if (flowStatus.contentEquals(okInputStatus)) {
                  foundOk = true;
            }
            if (flowStatus.equals(badInputStatus)) {
                  foundBad = true;
            }
      }

      if ((foundOk)&&(foundBad)) {
            outputStatus = mixedOutputStatus;
      } else if (foundOk) {
            outputStatus = okOutputStatus;
      } else if (foundBad) {
            outputStatus = badOutputStatus;
      }

            log.log(Level.INFO, "Decider setting exit status to "+out-
putStatus);

            // Note that this also because the Job Exit Status
            // Also note - no StepContext exists for a Decider

            return outputStatus;
     }

}
```

SamplePartitionMapper

```
package batch.samples;

import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Properties;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.batch.api.BatchProperty;
import javax.batch.api.partition.PartitionMapper;
import javax.batch.api.partition.PartitionPlan;
import javax.batch.api.partition.PartitionPlanImpl;
import javax.enterprise.context.Dependent;
import javax.inject.Inject;


@Dependent
public class SamplePartitionMapper implements PartitionMapper {

     private static final Logger log = Logger.getLogger( SampleParti-
tionMapper.class.getName() );


    @Inject
    @BatchProperty(name = "inputDir")
    String inputDir;


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

     /**
     * @throws IOException
      * @see PartitionMapper#mapPartitions()
     */
    public PartitionPlan mapPartitions() throws IOException {

     // Just use the default plan implementation
     PartitionPlan pp = new PartitionPlanImpl();

     // Get the list of files in the input directory
     File sourceFolder = new File(inputDir);
        File[] listOfFiles = sourceFolder.listFiles();
        LinkedList<String> ll = new LinkedList<String>();

        // Are there any files?
        if (listOfFiles != null) {
            for (int i=0;i<listOfFiles.length;++i) {
                if ((listOfFiles[i].isFile()) && (listOfFiles[i].get-
Name().endsWith(".txt"))) {
                // Add to our list of files to process
                    ll.add(listOfFiles[i].getCanonicalPath());
                     log.log(Level.INFO, "Found file ->"+lis-
tOfFiles[i].getCanonicalPath());
                }
            }
            // If we ended up with any files, create a partition prop-
erty for each one
            if (!ll.isEmpty()) {
                int fileCount = ll.size();
                Properties[] props = new Properties[fileCount];
                int partitionNumber = 0;
                Iterator<String> it = ll.iterator();
                while (it.hasNext()) {
                     String filename = it.next();
                     props[partitionNumber] = new Properties();
                     props[partitionNumber].setProperty("file-
name",filename);
                     ++partitionNumber;
                }

                // Set the property array into the Partition Plan
                pp.setPartitionProperties(props);
                // As many partitions as we have files
                pp.setPartitions(fileCount);
                // As many threads as we have partitions
                // Maybe this should be throttled somehow so we don't
ask for thousands of threads
                pp.setThreads(fileCount);

                pp.setPartitionsOverride(true);
            } else {
                // no files
                pp.setPartitions(0);
            }
        } else {
           // no files
           pp.setPartitions(0);
        }

        return pp;

    }
}
```

SampleSkippableException

```
package batch.samples;

public class SampleSkippableException extends Exception {

      private static final long serialVersionUID = 
5232667559006234399L;
      int badValue;

      public void badValue(int i) {
            badValue = i;
      }

      public int badValue() {
            return badValue;
      }

}
```

SampleSkipProcessListener

```
package batch.samples;

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

import javax.batch.api.chunk.listener.SkipProcessListener;
import javax.enterprise.context.Dependent;

@Dependent
public class SampleSkipProcessListener implements SkipProcessListener
{

     private static final Logger log = Logger.getLogger( SampleSkip-
ProcessListener.class.getName() );

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

     /**
     * @see SkipProcessListener#onSkipProcessItem(Object, Exception)
     */
    public void onSkipProcessItem(Object arg0, Exception arg1) {
     log.log(Level.INFO, "Handling skip value->"+((SampleSkippableEx-
ception)arg1).badValue());
    }

}
```

SampleListenerSetJobStatus

```
package batch.samples;

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

import javax.batch.api.listener.StepListener;
import javax.batch.runtime.context.JobContext;
import javax.enterprise.context.Dependent;
import javax.inject.Inject;

@Dependent
public class StepListenerSetJobStatus implements StepListener {

     private static final Logger log = Logger.getLogger( StepListen-
erSetJobStatus.class.getName() );

     @Inject
     private JobContext jobContext;


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

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

     /**
     * @see StepListener#afterStep()
     */
    public void afterStep() {
     String oldStatus = jobContext.getExitStatus();
     String status = new String("Step Listener Set Job Exit Status");
     jobContext.setExitStatus(status);
          log.log(Level.INFO, "Step Listener setting job exit status
from *"+oldStatus+"* to *"+status+"*");
    }

}
```

UserDataAnalyzer

```
package batch.samples;

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

import javax.batch.api.partition.PartitionAnalyzer;
import javax.batch.runtime.BatchStatus;
import javax.enterprise.context.Dependent;

@Dependent
public class UserDataAnalyzer implements PartitionAnalyzer {

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

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

     /**
     * @see PartitionAnalyzer#analyzeStatus(BatchStatus, String)
     */
    public void analyzeStatus(BatchStatus arg0, String arg1) {
     log.log(Level.INFO, "PartitionAnalyzer analyzeStatus:
BatchStatus="+arg0+" arg1="+arg1);
    }

     /**
     * @see PartitionAnalyzer#analyzeCollectorData(Serializable)
     */
    public void analyzeCollectorData(Serializable arg0) {
     String collectorData = (String)arg0;

     log.log(Level.INFO, "PartitionAnalyzer analyzeCollectorData:  da-
taWritten="+collectorData);

    }

}
```

UserDataCollector

```
package batch.samples;

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

import javax.batch.api.partition.PartitionCollector;
import javax.batch.runtime.context.StepContext;
import javax.enterprise.context.Dependent;
import javax.inject.Inject;

@Dependent
public class UserDataCollector implements PartitionCollector {

     private static final Logger log = Logger.getLogger( UserDataCol-
lector.class.getName() );


     @Inject
     private StepContext stepContext;

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

     /**
     * @see PartitionCollector#collectPartitionData()
     */
    public Serializable collectPartitionData() {
     String userData = (String)stepContext.getTransientUserData();
     userData = "Collector Data:  "+userData;

     log.log(Level.INFO, "PartitionCollector collectPartitionData, da-
taWritten = "+userData);

     return userData;
    }

}
```

UserDataPartitionedBatchlet

```
package batch.samples;

import java.util.logging.Level;
import java.util.logging.Logger;
import javax.batch.api.BatchProperty;
import javax.batch.api.Batchlet;
import javax.batch.runtime.context.StepContext;
import javax.enterprise.context.Dependent;
import javax.inject.Inject;

@Dependent
public class UserDataPartitonedBatchlet implements Batchlet {

     private static final Logger log = Logger.getLogger( UserData-
PartitonedBatchlet.class.getName() );

    @Inject
    @BatchProperty(name = "filename")
    String filename;

     @Inject
     private StepContext stepContext;

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

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

     /**
     * @see Batchlet#process()
     */
    public String process() {
           log.log(Level.INFO, "Processing file "+filename);
           stepContext.setTransientUserData(filename);
           stepContext.setExitStatus("Processing from "+Thread.cur-
rentThread().getId()+" completed successfully");
           return null;
    }

}
```

helpers/BatchletSetExitStatusToParm

```
package batch.samples.helpers;

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

import javax.batch.api.BatchProperty;
import javax.batch.api.Batchlet;
import javax.batch.runtime.context.StepContext;
import javax.enterprise.context.Dependent;
import javax.inject.Inject;

import batch.samples.BatchletSetExitStatusOk;

@Dependent
public class BatchletSetExitStatusToParm implements Batchlet {


     private static final Logger log = Logger.getLogger( Batch-
letSetExitStatusToParm.class.getName() );

     @Inject
     private StepContext stepContext;

    @Inject
    @BatchProperty(name = "useExitStatus")
    String useExitStatus;

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

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

      /**
     * @see Batchlet#process()
     */
    public String process() {
           log.log(Level.INFO, "Batchlet in step "+stepContext.getStep-
Name()+" setting exit status to "+useExitStatus);
           stepContext.setExitStatus(useExitStatus);
           return null;
        }

}
```

helpers/FakeFileProcessor

```
package batch.samples.helpers;

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

import javax.batch.api.BatchProperty;
import javax.batch.api.Batchlet;
import javax.enterprise.context.Dependent;
import javax.inject.Inject;

@Dependent
public class FakeFileProcessor implements Batchlet {

     private static final Logger log = Logger.getLogger( FakeFilePro-
cessor.class.getName() );

    @Inject
    @BatchProperty(name = "filename")
    String filename;

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

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

     /**
     * @see Batchlet#process()
     */
    public String process() {
          log.log(Level.INFO, "Processing file "+filename);
               return null;
    }

}
```

helpers/IntegerReaderState

```
package batch.samples.helpers;

import java.io.Serializable;

public class IntegerReaderState implements Serializable {
     private static final long serialVersionUID = 
3907191126941874291L;
     private Integer curInt;

     public IntegerReaderState() {curInt = new Integer(1);}
     public Integer curInt() {return curInt;}
     public void curInt(Integer i) {curInt = i;}


}
```

helpers/LoggingBatchlet

```
package batch.samples.helpers;

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

import javax.batch.api.Batchlet;
import javax.batch.runtime.context.StepContext;
import javax.enterprise.context.Dependent;
import javax.inject.Inject;


@Dependent
public class LoggingBatchlet implements Batchlet {

	private static final Logger log = Logger.getLogger( LoggingBatch-
let.class.getName() );

	@Inject
	private StepContext stepContext;


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

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

	/**
     * @see Batchlet#process()
     */
    public String process() {
    	log.log(Level.INFO, "Batchlet in control in step "+stepCon-
text.getStepName());
		return null;
    }

}
```

helpers/RandomDelayProcessor

```
package batch.samples.helpers;

import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.batch.api.chunk.ItemProcessor;
import javax.enterprise.context.Dependent;

@Dependent
public class RandomDelayProcessor implements ItemProcessor {

     private static final Logger log = Logger.getLogger( RandomDelay-
Processor.class.getName() );

     private Random rand = new Random();

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

     /**
     * @throws InterruptedException
      * @see ItemProcessor#processItem(Object)
     */
    public Object processItem(Object arg0) throws InterruptedException
{

     int delay = rand.nextInt(500);
     Thread.sleep(delay);  // nap up to half a second

 //        log.log(Level.INFO, "Processor delay = "+delay);

     return new Object();
    }

}
```

helpers/RandomDelayReader

```
package batch.samples.helpers;

import java.io.Serializable;
import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.batch.api.chunk.ItemReader;
import javax.enterprise.context.Dependent;


@Dependent
public class RandomDelayReader implements ItemReader {

	private static final Logger log = Logger.getLogger( RandomDelay-
Reader.class.getName() );

	private Random rand = new Random();
	private int itemsRead = 0;
	private int maxItemsRead = 100;

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

	/**
	* @throws InterruptedException
	  * @see ItemReader#readItem()
	 */
    public Object readItem() throws InterruptedException {

	Object retVal = new Object();

	int delay = rand.nextInt(500); // Random between 0-500
	Thread.sleep(delay);  // nap up to half a second

//		log.log(Level.INFO, "Reader delay = "+delay);

	// Something to stop the loop
	++itemsRead;
	if (itemsRead>=maxItemsRead) {
		retVal = null;
	}

//		log.log(Level.INFO, "Reader itemsRead = "+itemsRead);

	return retVal;
```

```
    }

      /**
     * @see ItemReader#open(Serializable)
     */
    public void open(Serializable arg0) {
    }

      /**
     * @see ItemReader#close()
     */
    public void close() {
    }

      /**
     * @see ItemReader#checkpointInfo()
     */
    public Serializable checkpointInfo() {
            return null;
    }

}
```

helpers/RandomDelayWriter

```
package batch.samples.helpers;

import java.io.Serializable;
import java.util.List;
import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.batch.api.chunk.ItemWriter;
import javax.enterprise.context.Dependent;

@Dependent
public class RandomDelayWriter implements ItemWriter {

     private static final Logger log = Logger.getLogger( RandomDelay-
Writer.class.getName() );

     private Random rand = new Random();

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

     /**
     * @see ItemWriter#open(Serializable)
     */
    public void open(Serializable arg0) {
    }

     /**
     * @see ItemWriter#close()
     */
    public void close() {
    }

     /**
     * @throws InterruptedException
      * @see ItemWriter#writeItems(List<java.lang.Object>)
     */
    public void writeItems(List<java.lang.Object> arg0) throws Inter-
ruptedException {

     int delay = rand.nextInt(500);
     Thread.sleep(delay);  // nap up to half a second

//         log.log(Level.INFO, "Writer delay = "+delay);
    }
```

```
	/**
	 * @see ItemWriter#checkpointInfo()
	 */
	public Serializable checkpointInfo() {
		return null;
	}

}
```

helpers/ThrowOnThirtyEightProcessor

```
package batch.samples.helpers;

import javax.batch.api.chunk.ItemProcessor;
import javax.enterprise.context.Dependent;


@Dependent
public class ThrowOnThirtyEightProcessor implements ItemProcessor {

     public boolean thrownOnce = false;

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

     /**
     * @throws Exception
       * @see ItemProcessor#processItem(Object)
     */
    public Object processItem(Object arg0) throws Exception {
     Integer i = (Integer)arg0;

     if ((i.intValue()==38)&&(!thrownOnce)) {
          thrownOnce = true;
          Exception ex = new java.lang.IllegalArgumentException();
          throw ex;
     }

     return arg0;

    }

}
```

JSL: CodingARetryableReader

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLoca-
tion="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/jobXML_1_0.xsd" id="CodingARetrya-
bleReader" restartable="true" version="1.0">
      <step id="Step1">
            <chunk>
                  <reader ref="batch.samples.RetryableIntegerSe-
quenceReader" />
                  <processor ref="batch.samples.helpers.ThrowOnThir-
tyEightProcessor" />
                  <writer ref="batch.samples.RandomIntegerWriter" />
                  <retryable-exception-classes>
                        <include class="java.lang.IllegalArgumentExcep-
tion" />
                  </retryable-exception-classes>
            </chunk>
      </step>
</job>
```

JSL: ExitStatusFlowControl

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLoca-
tion="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/jobXML_1_0.xsd" id="ExitStatusFlow-
Control" restartable="true" version="1.0">
     <step id="Step1">
          <batchlet ref="batch.samples.BatchletSetExitStatusOk" />
          <next on="OK" to="Step2" />
     </step>
     <step id="Step2">
          <batchlet ref="batch.samples.helpers.LoggingBatchlet" />
     </step>
</job>
```

JSL: JobContextTransient

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLoca-
tion="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/jobXML_1_0.xsd" id="JobContextTran-
sient" restartable="true" version="1.0">
     <step id="Step1" next="Step2">
          <batchlet ref="batch.samples.BatchletTransientSender" />
     </step>
     <step id="Step2">
          <batchlet ref="batch.samples.BatchletTransientReceiver" />
     </step>
</job>
```

JSL: SettingJobExitStatus

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLoca-
tion="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/jobXML_1_0.xsd" id="SettingJobExit-
Status" restartable="true" version="1.0">
     <listeners>
          <listener ref="batch.samples.JobListenerSetJobStatus" />
     </listeners>
     <step id="idvalue0">
          <listeners>
               <listener ref="batch.samples.StepListenerSetJobStatus"
/>
          </listeners>
          <batchlet ref="batch.samples.BatchletSetJobStatus" />
     </step>
</job>
```

JSL: UsingADecider

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLoca-
tion="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/jobXML_1_0.xsd" id="UsingADecider"
restartable="true" version="1.0">
     <split id="Split1" next="Decision1">
          <flow id="Flow1">
               <step id="Step1A">
                    <batchlet
                         ref="batch.samples.helpers.BatchletSetExit-
StatusToParm">
                         <properties >
                              <property name="useExitStatus"
value="OK"/>
                         </properties>
                    </batchlet>
               </step>
          </flow>
          <flow id="Flow2">
               <step id="Step2A" >
                    <batchlet
                         ref="batch.samples.helpers.BatchletSetExit-
StatusToParm">
                         <properties >
                              <property name="useExitStatus"
value="OK"/>
                         </properties>
                    </batchlet>
               </step>
          </flow>
          <flow id="Flow3">
               <step id="Step3A" >
                    <batchlet
                         ref="batch.samples.helpers.BatchletSetExit-
StatusToParm">
                         <properties >
                              <property name="useExitStatus"
value="BAD"/>
                         </properties>
                    </batchlet>
               </step>
          </flow>
     </split>
     <decision ref="batch.samples.SampleDecider" id="Decision1">
          <next on="OK" to="StepOK" />
          <next on="BAD" to="StepBAD" />
          <next on="MIXED" to="StepMIXED" />
     </decision>
     <step id="StepOK">
          <batchlet ref="batch.samples.helpers.LoggingBatchlet" />
     </step>
     <step id="StepBAD">
          <batchlet ref="batch.samples.helpers.LoggingBatchlet" />
     </step>
     <step id="StepMIXED">
          <batchlet ref="batch.samples.helpers.LoggingBatchlet" />
     </step>
</job>
```

JSL: UsingCheckpointAlgorithm

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLoca-
tion="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/jobXML_1_0.xsd" id="UsingCheck-
pointAlgorithm" restartable="true" version="1.0">
      <step id="Step1">
            <chunk checkpoint-policy="custom">
                  <reader ref="batch.samples.helpers.RandomDelayReader"
/>
                  <processor ref="batch.samples.helpers.RandomDelayPro-
cessor" />
                  <writer ref="batch.samples.helpers.RandomDelayWriter"
/>
                  <checkpoint-algorithm ref="batch.samples.SampleCheck-
pointAlgorithm" />
            </chunk>
      </step>
</job>
```

JSL: UsingCollectorAnalyzer

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLoca-
tion="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/jobXML_1_0.xsd" id="UsingCollecto-
rAnalyzer" restartable="true" version="1.0">
      <step id="Step1">
            <batchlet ref="batch.samples.UserDataPartitonedBatchlet">
                  <properties >
                        <property name="filename" value="#{partition-
Plan['filename']}"/>
                  </properties>
            </batchlet>
            <partition>
                  <mapper ref="batch.samples.SamplePartitionMapper">
                        <properties >
                              <property name="inputDir" value="#{jobParam-
eters['inputDir']}"/>
                        </properties>
                  </mapper>
                  <collector ref="batch.samples.UserDataCollector" />
                  <analyzer ref="batch.samples.UserDataAnalyzer" />
            </partition>
      </step>
</job>
```

JSL: UsingPartitionMapper

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLoca-
tion="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/jobXML_1_0.xsd" id="UsingParti-
tionMapper" restartable="true" version="1.0">
      <step id="Step1">
            <batchlet ref="batch.samples.helpers.FakeFileProcessor">
                  <properties >
                        <property name="filename" value="#{partition-
Plan['filename']}"/>
                  </properties>
            </batchlet>
            <partition>
                  <mapper ref="batch.samples.SamplePartitionMapper">
                        <properties >
                              <property name="inputDir" value="#{jobParam-
eters['inputDir']}"/>
                        </properties>
                  </mapper>
            </partition>
      </step>
</job>
```

JSL: UsingSkippableExceptions

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLoca-
tion="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/jobXML_1_0.xsd" id="UsingSkippa-
bleExceptions" restartable="true" version="1.0">
     <step id="Step1">
          <listeners>
               <listener ref="batch.samples.SampleSkipProcessLis-
tener" />
          </listeners>
          <chunk skip-limit="3">
               <reader ref="batch.samples.RandomIntegerReader" />
               <processor ref="batch.samples.RandomIntegerProcessor"
/>
               <writer ref="batch.samples.RandomIntegerWriter" />
               <skippable-exception-classes>
                    <include class="batch.samples.SampleSkippableEx-
ception" />
               </skippable-exception-classes>
          </chunk>
     </step>
</job>
```

JSL: UsingSplitFlow

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLoca-
tion="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/jobXML_1_0.xsd" id="UsingSplitFlow"
restartable="true" version="1.0">
	<step id="StepA" next="SplitBC">
		<batchlet ref="batch.samples.helpers.LoggingBatchlet" />
	</step>
	<split id="SplitBC" next="StepD">
		<flow id="FlowB">
			<step id="StepB1" next="StepB2">
				<batchlet ref="batch.samples.helpers.Logging-
Batchlet" />
			</step>
			<step id="StepB2">
				<batchlet ref="batch.samples.helpers.Logging-
Batchlet" />
			</step>
		</flow>
		<flow id="FlowC">
			<step id="StepC1" next="StepC2">
				<batchlet ref="batch.samples.helpers.Logging-
Batchlet" />
			</step>
			<step id="StepC2">
				<batchlet ref="batch.samples.helpers.Logging-
Batchlet" />
			</step>
		</flow>
	</split>
	<step id="StepD" next="FlowE">
		<batchlet ref="batch.samples.helpers.LoggingBatchlet" />
	</step>
	<flow id="FlowE">
		<step id="StepE1" next="StepE2">
			<batchlet ref="batch.samples.helpers.LoggingBatchlet"
/>
		</step>
		<step id="StepE2">
			<batchlet ref="batch.samples.helpers.LoggingBatchlet"
/>
		</step>
	</flow>
</job>
```

The Songs of Java Batch

When every post went up, I advertised it through LinkedIn. Somewhere around the 50th post, a song popped into my head that sort of went with the topic of the post. That started a tradition (which I came to regret a few times) of having a song (or two) associated with each post. There is a Spotify playlist with all the songs in it (Exploring Java Batch). Just to be complete, listed below are all the songs/artists I mentioned.

The End – The Doors
So Long, Farewell – Sound of Music
White Rabbit – Jefferson Airplane
Old and Wise – Alan Parson’s Project
Mr. Roboto - Styx
Juke Box Music – Kinks
Juke Box Hero - Foreigner
The Final Countdown - Europe
Psychobabble - Alan Parsons Project
How Many Friends – The Who
One in a Million – Guns N’ Roses
One Love - Bob Marley
Decision or Collision - ZZ Top
I Want a New Drug – Huey Lewis
Reelin’ in the Years - Steely Dan
Counting Flowers on the Wall – Statler Brothers
Empty Garden - Elton John
Be Prepared - Lion King
Ready for Love - Bad Company
Blowin' in the Wind - Bob Dylan
Rock Out, Roll On - Triumph
Feels Like the First Time - Foreigner
Which Way Do I Go Now - Waylon Jennings
What's Your Name - Lynyrd Skynyrd
I'm an old Cowhand - Gene Autry
Handle With Care – Traveling Wilburys
Hand Me Down World – The Guess Who
The Kids Are Alright - The Who
A New Day Yesterday - Jethro Tull
Get a Grip - Aerosmith
Talk Talk - Talk Talk
Think - Aretha Franklin
Dream On - Arrowsmith
Maps and Legends - R.E.M.
You've Got Another Thing Coming - Judas Priest
It's So Easy - Guns 'N Roses
Any Way You Want It – Journey
Money For Nothing - Dire Straits

The Midnight Special - Creedence Clearwater Revival
Can't You See - Marshall Tucker Band
Kiss On My List - Hall and Oates
Gangnam Style - Psy
Paint It Black - Rolling Stones
Breaking Up Is Hard To Do - Neil Sedaka
Karma Chameleon - Boy George
Uneasy Rider - Charlie Daniels
Simple Man - Lynyrd Skynyrd
Chain Gang - Sam Cooke
Chain of Fools - Aretha Franklin
The Chain - Fleetwood Mac
Fork in the Road - Neil Young
A Fork in the Road - Smokey Robinson
One Bourbon, One Scotch, and One Beer - George Thorogood
Join Together - The Who
We've Got Tonight - Bob Seger
Another Brick in the Wall - Pink Floyd
Separate Ways - Journey
Free Falling - Tom Petty
Touch and Go - The Cars
Love Me Two Times – The Doors
Living in the Past - Jethro Tull
I Guess I'll Have to Change My Plans - Tony Bennett
I Need a Plan - Nat King Cole
Get Back - Beatles
Got My Mind Set On You – George Harrison
Harden My Heart - Quarterflash
You Never Even Called Me by My Name - David Allen Coe
Feel Like a Number - Bob Seger
Days Are Numbers - Alan Parsons
Wrong Number - Dobbie Brothers
8675309 - Tommy Tutone
Step By Step - Eddie Rabbitt
Pick Up the Pieces and Go - Joe Bonamassa
Property - Todd Rundgren
Personal Property - Def Leppard
There's Your Trouble - Dixie Chicks
I Got Trouble - The Music Man
Nobody Knows the Trouble I've Seen - Sam Cooke
Too Much Time on my Hands - Styx
Unchained - Van Halen
Finish What Ya Started - Van Halen
Right Now - Van Halen
You Really Got Me - Van Halen
Take it to the Limit - Eagles

Call Me - Blondie
Check it out - Mellencamp
I Can't Drive 55 - Sammy Hagar
End of the Line - Traveling Wilburys
In The End - Rush
The End - Doors
Start Me Up - Rolling Stones
Come Together - Beatles
Changes - David Bowie
Listen to the Music - Doobie Brothers (used twice)
Break On Through - Doors
Party Like it's 1999 - Prince
Don't Stop Me Now - Queen
Eye in the Sky - Alan Parson's Project - used twice
Money - Pink Floyd
Wrap It Up - Fabulous Thunderbirds
Saturday Night's Alright - Elton John
Paperback Writer - Beatles
Return to Sender - Elvis
Double Vision - Foreigner
Come Dancing - Kinks
Lola - Kinks
Celebrate - Kool and the Gang
Who Are You - Who
With One Look - Andrew Lloyd Weber (Phantom of the Opera)
Shakedown - Bob Seger
Folsom Prison Blues - Johnny Cash
Bad Medicine - Bon Jovi
Life in the Fast Lane - Eagles
No Shirt, No Shoes, No Problem - Kenny Chesney
Urgent - Foreigner
Slow Down - Beatles
Southern Cross - Crosby Stills Nash
Born to Run - Springsteen
Share the Land - Guess Who
Blue Ridge Mountains - Larkin Poe
Where Was I - Kenny Wayne Shepherd
Why Am I Here - W.A.S.P.
Once in a Lifetime - Talking Heads
Big Tree, Blue Sea - Golden Earring
Rock'n Me - Steve Miller Band
Lumberjack Song - Monty Python
Working for the Weekend - Loverboy
Limelight - Rush
Choices - George Jones
Write This Down - George Strait

Walkin' After Midnight - Patsy Cline
Total Eclipse of the Heart - Bonnie Tyler
Maxwell's Silver Hammer - Beatles
Islands in the Stream - Kenny Rogers/Dolly Parton
I Walk the Line - Johnny Cash
Draw the Line - Arrowsmith
Cadillac Assembly Line - Joe Bonamassa
Telegraph Line - Schoolhouse Rock
Turn the Page - Bob Seger
Take This Job and Shove it - Johnny Paycheck
Carry On Wayward Son - Kansas
Sweet Dreams - Eurythmics
Memory - Cats (play)
Memory - Streisand
Changes - David Bowie
Try to Remember - The Fantasticks (play)
Smooth Operator - Sade
Killing Time - Triumph
Surprise, Surprise - Rolling Stones
Right Back Where We Started From - Maxine Nightingale
Won't Get Fooled Again - The Who
Gimme Three Steps - Lynyrd Skynyrd
Listen, Learn, Read On - Deep Purple
Fifty Ways to Leave Your Lover - Paul Simon
Control - Janet Jackson
Parallels - Yes
I'm Gonna Sit Right Down and Write Myself a Letter - Fats Waller
Back in Time - Huey Lewis
'round and 'round - Ratt
Listen to the Rhythm of the Falling Rain - The Cascades
Go Your Own Way - Fleetwood Mac

Document change history

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

March 1, 2022	Initial Version
March 1, 2022	Add document number

End of 6560420