WebSphere Application Server

Understanding Java Batch (JSR-352)

This document can be found on the web at:
www.ibm.com/support/techdocs
Search for document number **WP102706** under the category of "White Papers"

Version Date: March 20, 2018
See "Document change history" on page 33 for a description of the changes in this version of the document

IBM Software Group
Application and Integration Middleware Software

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

Many thanks go to Scott Kurz for answering all my obscure spec questions and Don Bagwell for all his help..with everything.

Contents

Introduction

JSR-352 defines the open standard for Java Batch. There is a very detailed specification available online that precisely defines the proper behavior for any JSR-352 implementation (of which there are several). The purpose of this paper is not to replace the specification. In fact, there will be many things covered careful in the specification which we will just gloss right over here. The goal of this paper is to simply convey the general idea of the Java Batch model in a more approachable style. If you read this document and find yourself asking "Well, what happens if…" you should go read the specification. Hopefully this document will raise those questions in your mind, because it means you've understood the basics enough to have complex questions. Enjoy.

Types of Steps

Batch jobs consist of steps and JSR-352 defines two types of steps: batchlet and chunk. In this section we'll take a look at both types and how and when you might use them.

Batchlet

Of all the fancy and complicated things you can do with a JSR-352 batch job, the batchlet is the simplest and the best place to start. A batchlet is simply a Java program that gets run for a step. The program gets control, does whatever it does, and then it ends and the step is over.

If you have written batch-like applications in Java that are run by starting up a JVM and then running your Java code then moving that application into a batchlet is usually pretty simple. Where your Java program probably had a main, your batchlet will have a `process()` method.

Parameters passed to your Java program will get handed to your batchlet a little differently (as we'll see later) but whatever processing you did you in your Java code can probably be done, possibly more easily, inside a batch (and maybe Java EE) container. You might copy a file from here to there or some other non-iterative processing.

A batchlet is a great place to invoke utility services. The WP102636 whitepaper has samples showing how to invoke DFSORT and IDCAMS from a simple batchlet.

The only extra duty a batchlet has to provide beyond whatever function it performs is handling a stop request. Through an operations interface you can request a job be stopped as it is executing. A batchlet step is notified of the request to stop by driving the `stop()` method on another thread. If your batchlet `process()` method is doing something that gets control occasionally (i.e. executing some sort of loop vs. just calling a utility and completing when it returns) then it can also check the state of some internal flag that your

`stop()` method can set. That allows the `process()` method to recognize a stop request and actually stop.

Chunk

For a batchlet your application code just gets control once and does whatever it needs to do. A chunk step is different because it always contains a loop. The batch container will create a loop around calls to your application. The idea is that your application will do some limited amount of processing, perhaps on a single item of data, and then return so the container can loop around and we do it again.

Obviously you could do this in a batchlet yourself, but the chunk model brings with it some pretty fancy options that the container takes care of for you. We'll dig into that as we work our way through the chunk programming model. But at a high level a chunk step is simply the container calling your application in a loop until you are done. In theory each pass through the loop does similar processing with different data, although really that is up to your application.

Fair enough, but we need more details. It turns out the application part of a chunk step is actually broken up into three main parts (instead of just one like a batchlet). In this section we'll take a look at those parts and then take a first look at the mysteries of checkpoint processing. For this section we'll assume everything always works…to keep things simple.

The Reader

The first interface you need to implement for a chunk step is an `ItemReader`. This class is, oddly enough, where you read the data being handled by the chunk step. Remember that the idea is that you're looping around reading different data and then doing similar processing on that data. In this part we're doing the read.

To implement a reader you need to know how to get to the data you are using as input to the processing part of the loop. It might be in a database table or a flat file or…well…anywhere. You don't need to know much in an `ItemReader`, but you have to know how to access your input data.

The first thing that will get control for an `ItemReader` in a chunk step is the `open()` method. This only happens once, outside the loop that forms the chunk step. This is your application's opportunity to open a file or establish a connection to a database or whatever it needs to do. You might even read a whole pile of data into memory here as a cache for later. Entirely up to you. The `open()` method gets control under a transaction which will be committed when all the open processing is complete. We'll talk later about other transactional considerations for the reader and cursor positioning for job restart. But we'll keep it simple for now.

The next method to implement is the `readItem()` method. This is where the action is for the `ItemReader`. You get to go actually read an item from wherever your application is

getting them from. You return that item to the caller. But how? The return type is just `Object` so it can be anything you like. You could just squish it into a `String` and return that. But this object is how you communicate with the processing part of the loop. Even if right now it seems like all you want to pass along is the data, somewhere down the road you're going to want to convey some other information besides the actual data you read. Probably best to create some sort of object that wraps the returned data. Then you can easily add other state data into that object.

What if your data is in some parseable format? Maybe it is a JSON string, or a blob of XML, or maybe a line from a CSV file, or even just some column delimited text…should you parse it? It might be nice to pull it apart and put it into nice handy attributes of your returned object. Processing code can then just pull the values it wants out of the object with appropriate getter methods.

That's especially handy if you are creating a generic `ItemReader` for this source of data that might get used with several different types of processing in different batch applications (woo hoo! Code reuse!).

On the other hand, parsing costs CPU cycles and elapsed time. And this code is called in loop, possibly a very very long loop. What if the processing code only needs one field out of the pile of fields in the record you read? Then you are wasting a lot of time each pass through the loop parsing data that nothing will look at. Multiply that little delay by the number of times through the loop and suddenly you've added hours to the execution time for the job (and a lot of CPU if you're being charged for it).

We'll get into this later, but it is possible to pass parameters into your `ItemReader` implementation so you could have the capability of parsing all the data but let the invocation tell you via parameters which fields (or sets of fields) to bother parsing out.

All the calls to `readItem()` occur inside a transaction that encompasses the contents of our loop. If your data access is transactional then it will participate in this transaction. This might be important because we commit the transaction occasionally in the course of processing (more on that later). If you have an open cursor on a database, that commit might close your cursor. To avoid that, try to keep your read-access to data out of the transaction by configuring the datasource as non-transactional, if possible.

The `readItem()` method is also the one in control of the loop. When it returns a null instead of some object representing what it read, that signals the batch runtime that you are all done and the loop will end.

Part of ending the loop is closing down the access to the data source. The `close()` method will be called at the end to take care of that.

There is one more method, `checkPointInfo()`, that we haven't talked about. We'll save that for later when we get to checkpoint processing.

The Processor

The processing part of our loop involves calling your implementation of the `ItemProcessor` interface. There is only really one method to implement, `processItem()`. It gets passed the object returned by our `readItem()` method just above. Reach into that and pull out whatever you need and off you go.

This is the meat of the batch application. Do calculations, drive business rules, sum, sort, or compute, whatever floats your boat.

Once you're done, you just need to decide one thing: what are you going to return? The processing part of the chunk step loop can return an `Object` to be passed on to the writer that we'll discuss next. It is entirely up to you whether you want to return anything or not.

If you return something it will make its way to the writer, and if not then the writer will never know about whatever you did here. Maybe that's ok. Maybe processing means looking for certain types of records and if this isn't one of those there is nothing to tell the writer.

But if you do have something, you need to fold it into an object to give to the writer. Again, this could just be a `String` or some other simple thing. But probably at some point in the future you are going to want to pass more information along so it is probably best to just wrap it in some class you can extend later on.

You should also note that the `ItemProcessor` is optional. You don't have to have one. If you don't, then the objects provided by the `ItemReader` are just passed straight to the writer we'll get to in a moment.

That's all there is for the processor. Even when things start to get complicated later on, the processor just does his one basic job. But it is the most important part of the step.

The Writer

The last major part of the chunk step is the writer, which has to implement the `ItemWriter` interface. This is where the results of the processing get handled. The name of the interface implies you are going to write results somewhere. The interface looks a lot like the `ItemReader` interface with the exception that the `readItem()` method is replaced by a `writeItems()` method (note the plural, it is important).

The writer starts, like with the reader, with an `open()` method. This gets called in the same transaction as the reader's open and is your opportunity to get set to write data wherever it is going to go. You might create an output file or open a connection to a database. The transaction wrapped around open processing is committed before we start into our loop for the first time.

Similarly, the `close()` method will be called along with the reader's `close()` after we have exited from the loop for the final time (when `readItem` returned a null). Do whatever is appropriate for wherever you are writing data.

The fun stuff is in the `writeItems()` method. As we noticed just above, the method name is plural. That's because it gets passed a list of items to write. But how can that be? The processor just returned one item (or maybe none). Where did we get multiple things to write? Ah…that's because the writer doesn't get called after every read and process! The batch container will loop around doing reads and processes until either the read returns a null and ends the loop or we hit a checkpoint. Ah that mysterious checkpoint that we've been hinting at. We'll get to it in just a moment, but for now ignore it.

Consider instead a very simple case where the reader just reads a few bits of data before declaring we are all done by returning a null. We've gone round the loop just a few times, say just five for example. And each time we went through our process code to have a look

Version Date: Tuesday, March 20, 2018

at the data. Each time process returned an object meant to go to the writer. But the batch container just looped around and did the read/process two-step again.

Well, now the reader has given us a null and we're all done. But we still need to write our results. Before the batch container commits the transaction that is wrapped around our loop, it calls our `writeItems()` method passing it a list of all the items returned by processing. In this case it will get a list of five items.

Now what? We write! But now we've got all five things to write instead of getting them one by one. This might make no difference at all, but for some datasources there might be ways to do bulk inserts where it is more efficient to insert five rows all at once rather than do them one at a time. That's why the loop is structured this way. It allows the writer to take advantage of those capabilities and write a bunch of items all at once. Write doesn't return anything when it is done.

The only other method to talk about here is the `checkpointInfo()` method. And just like we did under `ItemReader`, we're going to duck discussing that until we get to checkpoint processing.

Oh look…that's the next section! Finally….

Checkpoint processing

Let's start with what a checkpoint is. The idea behind a checkpoint is that you have reached some spot in processing where you want to remember that you got this far, just in case something bad happens. It is essentially the same as reaching a save point in a video game (if that analogy helps you). The batch job is about to confront the Boss Monster and if you die, you want to just start the game over right here and not have to start again from the beginning of the level.

In order to be able to start again from this spot, we're going to have to remember some things. What might we need to know? Well, we need to know where we are in reading records from our source. We might also have something we need to remember about where we are in putting results wherever the writer is putting them (maybe an offset into a file or something). And, perhaps most importantly, we want to be sure that everything we did so far (probably whatever the writer wrote) actually got written, because if we start over from here, we're not going to re-process those earlier records again. Back to our video game analogy, if you got the Super Weapon before you face the Boss Monster, you want to make sure you still have it when you start over from the save point.

How do we do this? Let's do the reader and writer first. As we mentioned earlier, both of those interfaces require you to implement a `checkpointInfo()` method. When we reach a checkpoint (when is that? Be patient..we'll get there) the batch runtime will call both the reader and writer and ask them for checkpoint information. This information will be placed into the Job Repository along with other information about this job, so we have it somewhere safe (well, as safe as your Job Repository is…).

What information should be provided as checkpoint info? Well, it depends on what your reader and writer will need to know if we have to start over again. Most simply this might just be a count of the record number that the reader was on, working through some known set of records. But what if the data source could change between now and when we try

Version Date: Tuesday, March 20, 2018

again? Well, you might need to remember more things about the data source. It depends on your data source. In creating your checkpoint data you have to consider all the things that might happen and what you would need to know if the job got restarted and this was all the information you had in order to properly pick up where you left off (if you can).

To help all this work, as we've mentioned earlier, the batch container wraps a transaction around the processing in our loop. Before we read our first record the transaction has started. When we reach a checkpoint, the container will let the writer do his thing, then collect checkpoint data from both the reader and writer and store that away in the Job Repository. Then the transaction is committed. That will commit the changes to the Job Repository and, at the same time, also commit changes made by the writer – assuming whatever the writer is doing will participate in the transaction. Just writing to a file won't honor the transaction and writes happen whenever they happen. But if you use a transactional JDBC datasource or some other transactional activity, then you can be sure the data you had written up to the checkpoint all happened.

We'll get to the mechanics of a job restart later. At this point we just want to understand the things we need to do up front to allow that to work.

Checkpoint policy

How often to checkpoints happen? You can trigger a checkpoint based on the number of records processed, the amount of time that has passed, or anything you else you can figure out.

To checkpoint based on the number of records, specify an item-count value on the chunk element in your JSL. This specifically controls the number of times the reader and processer will be called before the writer is called and a checkpoint taken.

You can also trigger a checkpoint on a time basis by specifying a time-limit value (in seconds). Then the read-process loop will continue until a process completes and we are past the time limit. Then the writer gets called and we do checkpoint processing. Note that we need to get through a read-process cycle and wind up past the timer interval, so the actual time of the checkpoint could be well after the timer has expired.

What if you specify both an item-count and a time-limit value? You get both! Whichever one happens first will trigger the checkpoint. Then both the counter and the timer are reset and we begin another checkpoint interval.

But what if what you want to do is complicated. Then you can specify `checkpoint-policy=custom` in the chunk definition and provide a `checkpoint-algorithm` element that points to a class implementing the `CheckpointAlgorithm` interface. This interface requires you to implement several methods.

To help you keep track of what is going on, there are `beginCheckpoint` and `endCheckpoint` methods that get control outside the loop (and outside the transaction).

And there is an `isReadyToCheckpoint` method that gets control after every item is processed. This method returns a `Boolean` that indicates whether now is the time to checkpoint or not.

Finally, there is the `checkpointTimeout` method which allows you to set a timeout value for the transaction that wraps each checkpoint. The value you can return here and how it all behaves gets rather complicated and interacts with various settings at the server level. There's an excellent writeup of those details here: http://stackoverflow.com/questions/36945450/how-do-i-configure-a-transaction-timeout-in-websphere-liberty-batch.

When Bad Things Happen

It's always something, whether it is bad input data, something configured improperly, bugs in the application code, somebody's bad assumption, or something else. Things go wrong. In this section we'll take a look at some different ways your batch application and the batch runtime itself can get involved in handling interesting situations.

Batch Status

The batch runtime itself will maintain a status for the job and for each step. You can access the current status using the `JobContext` or the `StepContext` (we'll get into a detailed discussion of these contexts a bit later on – for now just accept that they exist and your application can get to them).

The Batch Status will be a value from a spec-defined enumeration. The values are:

- STARTING
- STARTED
- STOPPING
- STOPPED
- FAILED
- COMPLETED
- ABANDONED

What good are these to a batch application itself? Not much. If you're actually inside the running job, the Batch Status is probably STARTED. This is probably more useful from an external perspective, looking at a job that has been submitted to find out what happened to it.

Exit Status

There is a separate status for each step and for the job itself that is entirely controlled by the application and is quite useful to the running batch application. This is called the Exit Status. It is a string and can be set to anything you like that you can get into a `String`.

You can set (and get) the Exit Status using methods on the previously mentioned Job and Step Context objects (which, again, we'll get to in more detail later).

After a step is completed, the Exit Status value can be used to determine what to do next. We'll have more detail on this later, but you can use the Exit Status to direct the flow of the job to an appropriate next step based on the results of the current step using the Exit Status value. You can also decide to end the job, successfully or unsuccessfully, using the Exit Status value.

For those familiar with z/OS JCL you can think of the Exit Status as something like the condition code for a step. In JCL you can have conditional execution of job steps based on the condition code of a previous step. One main difference is that JCL condition codes are numbers where the Exit Status is a `String`. That doesn't prevent it from being a numeric value in a `String` (i.e. "8").

Because you can use Exit Status for flow control it needs to always have a value after each step. And because the job itself needs some Exit Status value for the entire job it also always has to have a value. But what if your application doesn't set the Exit Status for each step (or for the job itself)? The batch runtime will take whatever the Batch Status value was for the step (or job) and set that as the value for the Exit Status.

Be careful with this… you might write JSL with flow control based on nice numeric Exit Status values but if you go down a path that doesn't set the step Exit Status you could wind up with a value of "COMPLETED".

It is also important to recognize that the Job Exit Status and each Step's Exit Status are distinct. There is no magic propagation of the last step's Exit Status to become the overall Job Exit Status. If you want an Exit Status value for the job itself, you have to use the Job Context to set it. And once it is set, that's the value for the job, even if it isn't done yet. If you have a job with a dozen steps and set the Job Exit Status in the first step, that's the Exit Status for the job unless one of the other steps sets a new value.

This means you need to have some awareness in writing each step as to whether or not you want to set an overall Job Exit Status or not based on what is happening. You might also want to set the Job Exit Status value from a listener, which we'll get to…soon.

Exception Handling

For a batchlet step it is pretty simple. If your batchlet throws an exception that it doesn't catch, then the step will fail. As we will see when we get to flow control, that doesn't necessarily mean the job is over. But the step fails and that's the end of the step.

For a chunk step the rules are more complicated. If you know what exceptions your application is likely to throw in situations where you want a chunk step to keep going, you can add extra stuff into the JSL to handle it. That's pretty vague. What are we talking about?

Skippable Exceptions

Well, suppose your chunk step `ItemReader` is reading records from a database. Suppose each record consists of a bunch of columns that the `ItemReader` establishes as attributes of an object that will be passed to the `ItemProcessor`. Suppose along the way the `ItemReader` does some basic validation that the column values contain reasonable

values. Maybe the customer's year of birth is contained in one column and the application validates that the year isn't in the future.

Alright…that seems reasonable. So what should the `ItemReader` do if it finds a row with a customer born 100 years from now? Well, we don't want to just process this record like any other. Something is wrong. But we don't want to fail the entire job because of one bad record.

The first thing we need to know to handle this is what exception the application is going to throw when it encounters an error in the record data. For simplicity's sake, let's say it throws an application-defined `BadCustomerRecord` exception. (A real exception probably has a package name…).

In our JSL for this step we can indicate that we know we might get a `BadCustomerRecord` exception and we just want to skip this record and move on. Skip processing basically says to ignore this record and just call the `ItemReader` again to go read another record. You code that up by specifying a `skippable-exception-classes` element in the JSL for the step. Inside that element you can specify a list of exception classes that you are expecting and want to just skip the record.

You can both include and exclude exception classes from skip processing and the rules for handling exceptions that are in both lists or exceptions that are extensions of other specified exceptions get complicated. Read the spec if you want to get clever.

But what happens to the record we skipped? There's a listener that can get control for this case and we'll cover listeners in the next section.

What if every record is bad? Or a lot of them? We don't want to skip all the input and declare success. As part of the chunk element configuration you can specify a `skip-limit` which is the maximum number of skippable exceptions to allow before failing the step.

Retryable Exceptions

Is skipping the record the only thing you can do? Nope. You can also retry. Let's take a different example. Suppose your `ItemProcessor` gets passed information about a record that the `ItemReader` acquired somehow and it is possible that something might go wrong processing the record. There's nothing wrong with the data in the record itself, but maybe there's some external resource the processor needs to access in order to do whatever it does and sometimes that goes all funky. In that case the `ItemProcessor` throws some particular exception that indicates some sort of transient error occurred processing this one record. What we'd like to do is just try this one again. There's a way to do that.

Just like the skippable exceptions we talked about earlier, you can also include (and exclude) exception classes in a `no-rollback-exception-classes` element in the JSL for the step.

If the processor throws an exception that matches an entry in this clause the batch runtime will just call the `ItemProcessor` with the very same object it just used. The processor gets a second try at processing this record.

And just like skipped records, you can code a `retry-limit` in the chunk JSL element to stop the retry processing if it happens too many times.

And just like skipped records, you can specify a retry listener that will get control to do any special processing required (like generate an email to the owner of the resource to complain about it going all wonky again).

Retry Rollback and Some Complications

There's yet another exception handling option called `retryable-exception-classes`. Exceptions listed in this clause are also handled via retry processing, but the whole chunk will be retried. The reader and writer close methods will be called, the transaction wrapping the chunk processing will be rolled back, and we start again at the beginning of the chunk. This is processing you might want in the case where something more serious has gone wrong.

There's an interesting twist to the retry-with-rollback processing. When processing begins again at the start of the chunk, the specification requires that processing proceed as if the chunk limit was an `item-count` set to one. This happens regardless of how your JSL specifies the chunk limit (`item-count`, `time-limit`, or `checkpointAlgorithm`). That means that the loop will call the reader, processor, and writer and then commit the transaction in a one-by-one fashion. The question is, when does that one-by-one processing stop? The specification isn't really clear about it, but it appears to stop when processing gets past the record that caused the exception before resuming normal checkpoint intervals.

For example, suppose you were processing a chunk step with an `item-count` of ten and a retryable exception occurred processing item number five. The transaction would roll back and we start again with item number one. Processing continues, one-by-one, through item number five. Then we go back to processing items in ten-item chunks as before. If you are counting checkpoints in a job and expect a certain number to occur this might surprise you. Maybe you know there are 1000 records and you checkpoint every 100 records, so you expect 10 checkpoints. But if a retryable exception occurs part way through one set of 100 records you could easily end up with 50 checkpoints processing the step (the expected ones plus a bunch processing records one-by-one recovering from a retryable exception).

Furthermore, we've used some fairly simple examples of scenarios where you might want to skip or retry and that has allowed us to avoid some interesting complications.

For example, suppose the `ItemReader` throws some exceptions that are skippable and other exceptions that are `retry-no-rollback` exceptions. For the processor the `retry-no-rollback` exception just drove the processor with the same object to process. But what should a reader do? For a skippable exception it needs to move on to the next record, but for a `retry-no-rollback` exception it needs to try to re-read the same record. How does it know which case this is?

There are a few different ways to handle this. The simplest approach is to just have the `ItemReader` understand how the JSL will handle different exceptions and, before it throws

the exception, remember in the object state what happened and do the right thing when it gets control to read an object again.

That seems a bit tacky though. Another approach is to exploit the skip and retry listeners that we haven't talked about yet. They will get control appropriately depending on what processing is occurring and could communicate with the `ItemReader` to let it know what to do next. Maybe the `ItemReader` is keeping track of what record number it is currently on (as part of the checkpoint data) and the listeners could 'adjust' the current record value.

Or something else.

Exceptions during write processing are interesting also. A `retry-no-rollback` exception will just call the `ItemWriter` again to try again to write the list of objects. If the writer managed to write some but not all of the objects you need to be careful to not write them again.

A `retry-with-rollback` exception will just retry the whole chunk, so that's the same processing as in any other processing in the chunk.

A skippable exception on the `ItemWriter` is interesting because we will just skip the writer and move on..to commit the transaction and whatever it managed to write before the exception occurred. I'd be very careful with skippable exceptions from the writer.

Listeners

Listeners are special classes that allow the application to have code that gets control at different points in the processing of the job. Some of these points are just at convenient times, like before and after a step executes. Some of them allow you to get control when interesting things happen, like an uncaught exception from other application code.

Listeners are defined at the job level and at the step level. Each step can have its own set of listeners, separate from the listeners used in other steps.

When you define a listener in the JSL for a job, you simply include the listener element at the right level (the job, or inside the step you want listeners for) and list the listener classes. You don't have to tell the JSL what the listener is and when it should get control. The batch runtime figures it out from the interface the class implements.

Job Listener

There is only one listener you can define at the job level, and that is, amazingly, the Job Listener. This one is pretty simple. It allows you to get control at the beginning of the job and at the end of the job. That's it. A chance to set something up or to clean up something. This might also be an excellent place to set the Exit Status for the Job. It is the only place that is guaranteed to get control at the end of the job, regardless what step sequence occurred.

Step Listeners

There are a lot of different listeners you can have inside a step. The simplest is the Step Listener itself. Like the Job Listener, it gets control before the step and after the step.

Getting control before the step actually starts allows you to do initialization or setup of whatever you might need for the step. Getting control after the step can allow cleanup processing, but also might be a good place to set a final Exit Status for the step. You might set that somewhere along the way as processing takes its course, but the Step Listener's `afterStep()` method is a chance to take a last look and make a final decision about how this all actually went.

For a Batchlet step the Step Listener is the only listener that applies. That's important because if something goes wrong in a Batchlet and an exception is thrown that the Batchlet itself doesn't catch, the Step Listener's `afterStep()` is the only opportunity you have to sort things out. The Step Listener can examine the step's Exit Status and determine that something went wrong. You can leave the Exit Status as the Batch Status of FAILED or customize it to something more specific and helpful, if you can tell.

For a Chunk step the Step Listener still gets control before and after the step, but you have a lot of other options.

Chunk Step Listeners

Once we are inside a Chunk step there are more listeners that can get control. These occur 'inside' the Step Listener's before/after step methods.

Chunk Listener

The first of these is the Chunk Listener itself. Like the Job and Step Listeners it has methods that get control before and after processing, in this case for each chunk. Remember that a chunk is the read/process loop that ends with a write when we've hit the end of a chunk as defined by a count, a time, or a checkpoint algorithm. The before and after methods get control before the first read and after the write but before the transaction is committed. Mostly.

The Chunk Listener has a third method called `onError()`. This method gets control if an exception is thrown during the processing of the chunk that isn't handled in some way. The `onError()` method will get control instead of the `afterChunk()` method and will be given the exception that was thrown as a parameter.

Read/Process/Write Listeners

There are also listeners that get control during the processing of each part of the chunk loop processing. For the reader there is a Read Listener, for the processing there is a Process Listener, and for the writer there is a Write Listener.

Each of these Listeners has its own before and after method that get control before and after each part of the processing. So the Read Listener can get control before and after each call to the `ItemReader` to read a record. Likewise the Process and Write Listeners can get control before and after the call to the `ItemProcessor` and `ItemWriter`.

Each Listener gets information pertinent to what it is doing or has done. In the case of the Read Listener, the `afterRead()` method will be passed the `Object` that was read. The Process Listener's `beforeProcess()` also gets the `Object` that was read and the

`afterProcess()` method gets both the read `Object` and the `Object` (if any) that was the result of the processing. For the Write Listener both the before and after methods get the List of `Objects` that were provided to the writer.

All of these control points allow Listeners to observe, and perhaps log, any interesting information that comes along. They can also meddle with the contents. Perhaps you are using a generic reader to read from some data source, but the Processor needs some version of the data that has been manipulated in some way. You could use either the Read Listener's after method or the Process Listener's before method to adjust the read `Object` accordingly. Similarly with the edge between processing and writing. The Listeners give you the opportunity to help 'glue' together what might be standard readers/processors/writers by adjusting for differences between them.

All three of these Listeners also have an `onError()` method that will get control in the event of an exception. The Listeners are all provided the exception that was thrown. You cannot 'handle' the exception in the Listener. That has to be done with the skip/retry JSL exception lists we have discussed earlier. The `onError()` method does allow you to log something about the error. For example, if the Reader didn't like the format of the record it read, it might throw an exception and include the content of the bad record in the exception class thrown. The Read Listener could then extract that data and log it somewhere as a record that needs to be cleaned up or otherwise handled specially.

Skip and Retry Listeners

As if that wasn't enough Listeners, there are actually still SIX MORE. There are additional Listeners that get control for Skip and Retry processing and for each of those there is a Listener for the read, process, and write phases.

There is a Skip Listener for Read which gets the exception thrown by the read that will cause a skip. The Skip Listener for Process gets both the exception thrown by processing and the `Object` that was being processed. The Skip Listener for Write gets passed the entire List of objects to be written as well as the exception thrown while writing.

Remember that this is how the application processing knows that skip processing is occurring. This is your chance to communicate with the relevant Reader, Processor, or Writer that this item (or items for Write) should be skipped and to adjust accordingly.

Similarly, there are Retry Listeners for Read, Process, and Write. They all receive the same parameters as described above for the Skip Listeners. And again, this is your chance to tell the relevant Reader, Processor, or Writer what to do. For example, in the case of the Reader, the Skip Listener needs to make sure the next call to `readItem()` will read the next item, while the Retry Listener needs to make sure the next `readItem()` tries to re-read the item.

Remember that Retry processing can just retry the one item or it can retry the entire chunk, depending on whether the exception was in the no-rollback exception list or the retryable exception list. The Retry Listeners don't get told what is happening, so your application just has to know from the exception that was thrown how the JSL was coded to handle it and set things up properly. It can get a bit tricky.

Version Date: Tuesday, March 20, 2018

Listener Multiple Inheritance and Multiple Listeners

One more item about Listener implementations. We noted earlier that in the JSL you simply define a Java class as a Listener and the batch runtime will figure out what Listener Interface it implements and give it control appropriately. What if a single Java Class implements multiple Listener Interfaces? Suppose you have a single class that implements both the Step and Chunk Listener Interfaces? The method names are all different so there won't be any confusion there. But which Listener is registered and given control? The answer: both of them.

But what if my JSL defines two Listener classes and both of them implement the same Interface? What happens then? Well, the spec says that you can do that. And both Listeners will get control. If you specify two Step Listeners both will get control before and after the Step. The spec specifically says that you can't count on what order they will get control.

That said, it might make some sense to put the implementations of multiple Listeners in a single class. Maybe the Skip and Retry Read Listeners go together. But don't get carried away. Do things that make sense.

Flow Control

We've talked a lot about what goes on inside an individual step, but only hinted a bit at how you get from one step to another. In this section we'll take one step away from the details of the JSL in the step and look at the overall flow of a job.

The Basics

There are three basic rules for flow through the steps of a job.

1) The first step in the JSL file is always the first step executed
2) The currently executing step must indicate what the next step should be
3) If no next step is indicated the job finishes

That's it. You start at the top and run the first step and then go wherever it tells you. After that the indicated step is run and when it is done you go wherever it says to go. If you ever reach a step that doesn't tell you what step to go to after that, then you are done.

The simplest thing to do is have each step unconditionally proceed to another named step. You do that in the step element by specifying the `next` attribute and the ID of where to go next. That next thing is probably another step, but can be some other special things that we'll get to.

The only other thing to remember about basic flow control is that you are NOT allowed to create a loop. If job execution winds its way back to a step that it has already executed the job will fail.

Conditional Execution

Basic flow control with each step specifying an unconditional next step is easy, but pretty unrealistic. Things go wrong. At the very least your JSL should be prepared for things to go wrong. It is very rare that no matter what happens in STEP1, you always go to STEP2.

You can introduce conditional flow control through various elements that are included underneath the step element. These elements, while included under the step, are not technically considered to be part of the step itself and are only evaluated after the step's processing is completed (not the same, necessarily, as having the step's Batch Status set to COMPLETED).

There are four special elements you can include to manage flow control and you can specify each of them multiple times. They are evaluated in the order they occur in the JSL. That will be important.

All four elements begin by letting you specify an Exit Status value to match. Essentially you are specifying "if the Exit Status is XYZ then do this, else if the Exit Status is ABC then something else, etc."

Next

The first transition element is `next`. In this case you simply specify the Exit Status you wish to match (wildcarding is allowed) and then, just like in the `next` attribute of the step element, the ID of where you want to go next. This is pretty normal flow control. If the step completed with this Exit Status, go here, if it completed with some other Exit Status, go there. Nice, simple conditional flow control.

Fail

But what if something really bad happened? You don't want to run any other steps. You want to stop right here. Well, you can manage that by just not providing any flow control and the job will stop. But the job will be considered to have completed successfully (Batch Status of COMPLETED). That might not be what you want. Something went wrong. We'd like to make it look that way.

For that you use the `fail` element. Just like the `next` element, `fail` allows you specify an Exit Status you want to match (again, wildcard if you need to). If the Exit Status from the step matches, then the job execution ends with a Batch Status of FAILED. What about the Exit Status? We already have the step Exit Status, but we should set the job's Exit Status too (remember they are separate). The application could set that, but you might want to have it here in the JSL. So the fail element allows you to specify an Exit Status value for the job here too.

Stop

If you want to stop execution of the job due to an error (or whatever reason) use the `stop` element. Just like `next` and `fail`, you specify an Exit Status (wildcarding if you must) to match. And you also get to specify an Exit Status value to be used for the job's Exit Status. That's exactly the same as the `fail` element. What's different is that the Batch Status of

the job will be STOPPED. With `stop` you also get to specify the id of a step to restart at if the job is restarted. We'll get more into job restart processing later.

End

The final flow control element is end. This is for the case where you are perfectly happy to end the job right here. Nothing went wrong, you just want to set a final job Exit Status and mark the job COMPLETED. That's the difference from just walking off the end of the JSL by leaving the batch runtime with no next step. An end element is a nice tidy completion of the job with a JSL-coded job Exit Status. You code it just like the fail element by specifying an Exit Status to match (wildcarding supported, of course) and the job Exit Status you want set.

Ordering

As we said earlier, these flow control elements are evaluated in the order they are found. If you code specific Exit Status values to match then it doesn't really matter. But if your application is handing back interesting strings (vs nice simple numbers) as Exit Status values and you are having a good time with wildcards in the matching Exit Status values, be careful to not accidentally match something you didn't mean to match.

Split/Flow

JSR-352 contains a special job flow construct that allows concurrent execution of different steps. This is a great way to get some parallelism in your job and maybe cut down on the overall elapsed time. But you have to be really sure those concurrent steps don't really have dependencies.

The Flow

The first piece to understand is the flow. A flow is a grouping of steps. You can have a flow all by itself (without the split stuff). This allows you to treat a group of steps as if it was a single thing. The flow has an ID so it can be the target of flow control statements. For example, when an error occurs there might be a whole sequence of things you need to do. You could group those steps together into a flow and any `next` elements that need to do error processing could point to the entire flow instead of just whatever happens to be the first step in the sequence.

In some ways you can consider a flow to just be a giant step. The flow itself can have a `next` attribute that tells the job where to go, unconditionally, when the flow ends.

If you look at the specification it would appear that you can also have the four control elements we discussed earlier (`next`, `fail`, etc) as child elements of the flow and conditionally route control based on the results of the last step in the flow. But if you discover Section 8.9.5 you'll see that isn't the case. The syntax of JSL will let you do that, but the specification makes clear that you shouldn't. That's because there are some messy situations where it isn't clear what the Exit Status of the 'last step' actually would be.

It is important to note that all next attributes or elements for steps within the flow must direct control to another step within the flow. You cannot branch out of the flow. You also should not branch from a step outside the flow to a step inside the flow. The elements inside a flow should be considered 'invisible' externally.

From the outside then, you can reference the flow as a complete entity, branching to it as if it was a step. The flow itself can unconditionally direct control just like a step. So, in a sense, it is just like a giant step made up of internal steps.

The Split

A split is another special JSR-352 construct. A split has an ID just like steps and flows, so other steps can direct control to the split on a `next` or other control elements.

A split specifies an unconditional `next` attribute. This indicates where control should proceed when all the processing in the split is complete. You cannot have conditional controls directly as part of the split (we'll see how this is handled in a moment). If the split does not specify a `next` attribute, the job is complete when the split completes.

The only inner elements allowed within a split are `flow` elements. Thus a split is made up of one or more flows. Each flow will be executed on separate threads (not the thread where the split got control). You can have as many flows inside a split as you like. A flow can contain another split with more inner flows.

If you think you can make use of a split/flow in your job, first group together steps in the job into flows and then determine if those flows can execute concurrently or not.

The Decider

When all the flows within a split complete, what do we do next? We saw just above that the only control statement you can make is to indicate a next step after the split completes. But a lot probably happened inside there. You may need to make a complicated decision about what to do next. That's what the `decider` is for.

The d`ecider` is a special kind of step. It has an ID (so the split's `next` attribute can point to it) and it can have all the usual control elements based on Exit Status values.

But what does it do? There is an interface called a `Decider` that you implement with a single method called `decide()`. That method gets passed an array of `StepExecution` objects representing the results of whatever ran before it.

In the case of a decider following a split, the array contains the `StepExecution` objects for the final step of each flow that was part of the split. From there you can get to the Exit Status of the steps (among other things). The decider uses that information to determine its own Exit Status to be used to, ahem, decide where to go next.

The `String` returned from the `decide()` method is the Exit Status used in any conditional flow control elements that you define as part of the decider step. It also becomes the overall Exit Status for the Job itself.

You can use a decider as we have discussed here to resolve the flow of control after parallel processing of flows within a split. But you can place a decider element anywhere in

the processing of a job. The decider always receives the `StepExecution` object for whatever came before it and can use that to set a job-level Exit Status as well as make a decision about what the job should do next.

Parameters, Properties, and Contexts

Contexts

We're going to take these in reverse order. We've talked about Contexts a bit before this. You can access the Job or Step Context to set the Exit Status for the job or step. But how? What do you have to do to get to those contexts?

It is done through injection. There are several different technologies that can be used to make that work and how it is done depends on what implementation of JSR-352 you are using. We won't get into the mechanics of it here, just assume it works.

In your application when you need to access a context you just put an `@Inject` on the line before your declaration of the `StepContext` or `JobContext`. The result will be that your local Job or Step Context variable will be set to point to the appropriate batch artifact.

JobContext

What can you do with the `JobContext`? We already talked about the ability to get and set the Exit Status for the job. You can also peek at what the batch runtime has set the Batch Status value to (probably STARTED). You can get the identifiers for the job (instance and execution) as well as the jobname (from the JSL).

You can also access the Job Properties. Most JSL elements allow you to specify properties and we'll talk about how those get injected in to the related application artifact later. But there is no 'job' artifact for job-level properties so you have to access them through the Job Context. As we'll see job properties might have values from the JSL or might have values provided by the submitter of the job. If you need to access any of those then the `JobContext` is the way you get there.

The last thing of interest in the `JobContext` is the Transient User Data. This is pretty much just what it says it is. The application can use the `JobContext` to set any `Object` into the Transient User Data and to get it back out again.

The Transient User Data is, like the name says, transient. So it won't survive a job restart. And there are some questions about how the data is handled in multi-threaded jobs and other situations. Behavior may vary a bit between JSR-352 implementations so be careful how you use it.

StepContext

The `StepContext` is, obviously, similar to the `JobContext` but for the current step in execution. It allows you to access the step execution ID and a Properties object containing step-level properties from the JSL. You can also get the step name (from the JSL). And,

as with the `JobContext`, you can get the Batch Status and get and set the Exit Status (for the step).

The `StepContext` also provides access to the same Transient User Data discussed above. There is also a method to access the last exception thrown from application code within the step that was caught by the batch runtime. This might prove useful in some of the error handling cases we've discussed earlier.

You can also access the step `Metrics`. The `Metrics` object contains counts of some things that are interesting when processing a chunk step. The `Metrics` include counts of the number of reads and writes and the number of skips and retries.

Finally, the `StepContext` provides access to the Persistent User Data. This is different from the Transient User Data because it persists. This data must be serializable and is associated with this particular step. It is saved (persisted) at checkpoints and the end of the step. It will be available if the step runs during a job restart. When we get to partitions we will have to be careful with the Persistent User Data.

Properties

I think every artifact you can specify in JSL can have properties associated with it. Those properties are then available to the artifact when it is given control. You can set properties on a batchlet, on a reader, on a listener, even a decider. When your application code is driven, the value of those properties are available.

All the properties are just name/value pairs of strings. The name is a string and the value is a string. To get the value of the property into your application code, you need to know the name. In your code you specify the `@Inject` like we did for the Contexts, but then you follow that with `@BatchProperty` and `name="property_name"` in parenthesis afterwards. This tells the injection code to inject the property with the specified name. Following that you declare the String variable that you want the value of the property placed in. For example:

```
@Inject
@BatchProperty(name="myproperty")
String myproperty;
```

Will take whatever the value of the `myproperty` property is and place it in the local `myproperty` variable. You can specify and access any property from the JSL inside your application code. Be warned, you can't access injected property values from the constructor. The object has to exist first.

What if you try to inject a property that isn't in the JSL? It won't get a value, so the variable will be null. Do you have to inject all the properties from the JSL? No.

Parameters

As we've mentioned earlier, the submitter of the job can provide parameters that can be used in the processing of the job. How? Where do these parameters end up? The only place you access them is through substitution into property values. For any property value

you can code a specific value, allow the value to be taken from the value of a job parameter, or use a default value if the job parameter wasn't specified when the job was submitted.

The syntax is a bit tricky. Let's start with a simple property:

```
<property name="propertyName" value="propertyValue" />
```

Now suppose we want the submitter to be able to supply a job parameter to set the value of `propertyName`. As a general rule the parameter name should probably match the property name to avoid confusion. But, in this example, we'll ask the submitter to supply a parameter called `parameterName` with some value. In that case the property syntax becomes:

```
<property name="propertyName" value="#{jobParameters['parameterName']}"/>
```

Say what? Ok, the `#{…}` bit means we're going to substitute in something here and not just have a literal string value. You can substitute in from job parameters (as we're doing here) or job properties (so a step can get a property value from another property) or even from system properties. So the pound sign and curly brackets say we're substituting, and the `jobParameters` tells us to get a value from a job parameter, now we just need to know which parameter. The name of the parameter goes inside the square brackets, inside single quotes. Because it does.

Well that's all well and good as long as the submitter of the job gives us a value for `parameterName`. If he doesn't and our application injects the value of `propertyName` somewhere it will get a null. Also fine as long as we're expecting that.

But maybe we'd like to supply a default value just in case. That way the application code always gets a value. In our case, let's supply a default value of `VAL`. To specify a default you put a question mark followed by a colon and then the default value, followed by a semi-colon. This goes outside the curly braces but inside the double quote. Like this:

```
<property name="propertyName" value="#{jobParameters['parameterName']}?:VAL;"/>
```

If you want to get clever, the default value could also be a value substituted from another value (e.g. use the #{…} business in the default value part of the string). As with most other things, there is a lot of opportunity to get very clever and create problems. Remember Brian Kernighan's rule about clever programming….

Partitioned Steps

Earlier we discussed the split/flow construct that allowed us to run different steps concurrently as part of a single job. A partitioned step allows us to run multiple copies of the same step, concurrently. For example, suppose we have a chunk step that is going to process a million records. If we know there aren't any contention issues, we might like to run 10 copies of that step, each processing a different range of 100,000 records. In theory, that might allow the step to finish in less elapsed time. Of course it depends a lot on what processing is going on and how locking is handled etc.

How do we make this happen? First we need to be able to specify in the JSL that we want the step to run partitioned. That's easy enough. We just specify `<partition>` as an element of the `<step>`. Note that the partition element is a peer to the `chunk` or

`batchlet` element. We're going to put information inside the `partition` element that explains how the partitioning is going to work.

But wait… did we just say batchlet? A partitioned chunk step makes sense. Just see the example above. But why would you want a partitioned batchlet? Why not? A batchlet is just some piece of code that runs as a step. You might want to run multiple copies of that at the same time doing similar but slightly different things. Suppose your batchlet copies a file. You could run ten copies of it concurrently to copy ten different files at the same time.

All this leads us to the main questions with partitioned steps. How do the partitions know what to do? How does each copy of the chunk step know what range of data to work with? How does each copy of the batchlet know which file to copy? We need a plan. In fact, we need a `partitionPlan`!

Partition Plan

The first thing we need in our plan is how many partitions we're going to have. We also get a chance to tell the batch runtime how many threads we'd like to use. You might have ten partitions but know that running more than five at a time creates too much contention for some resource. There's no guarantee you will actually get five running at a time, we are just giving the runtime some guidance. You do all this with the plan element:

```
<plan partitions="10" threads="5" />
```

That's great, but how will those ten partitions know what to do? We need to give them all their own properties! The idea is that each partition, being just a copy of the same code, will be prepared to inject properties that tell it what to do. For example, our batchlet would expect to get a property injected that tells it the name of the file to copy. That might be a property named `sourceFileName`. We just need our `partitionPlan` to specify ten different values for `sourceFileName` and the runtime will take care of giving each copy of the batchlet the right value.

To do this we set up properties blocks underneath the `partitionPlan` element. Each `properties` block specifies the number of the partition it belongs to. Partition numbers start at zero. We might have something like this (cutting down to two partitions for simplicity):

```
<plan partitions="2" threads="2" >
     <properties partition="0" >
          <property name="sourceFileName" value="file1.txt"/>
     </properties>
     <properties partition="1" >
          <property name="sourceFileName" value="file2.txt"/>
     </properties>
</plan>
```

That seems pretty straight-forward. But presumably those properties are going to be used by the application. How do you specify the property to be used? Well, there's a trick to the

property substitution we talked about earlier (under job parameters) that you can use with partitions. Our batchlet might have a property like this:

```
< properties >
      <property name="source" value="#{partitionPlan['sourceFileName']}"/>
</properties>
```

That's the same `#{…}` syntax we used before, but in this case instead of substituting in from a job property, we substitute from the partition plan. The batch runtime will take care that each partition (each copy of the batchlet in this case) gets the property value injected from the properties for that partition in the plan. So in our case partition zero gets 'file1.txt' substituted in and injected into the batchlet and partition one gets 'file2.txt'.

But this requires you to code in the JSL exactly how you want the partition parameters to work. That might be easy in a case where you know parameters as in our case where we have some files to copy. But what if you don't know? What if you needed to go look in a directory and find all the files? Or for a partitioned chunk step you might need to find out how many records there are to process and decide on the fly how to partition it? For that, we use a `partitionMapper`!

PartitionMapper

Instead of specifying a `plan` element inside the `partition` you can specify a `mapper` instead. The `mapper` references another batch artifact which implements an interface called `PartitionMapper`. Like every other artifact, the mapper can be provided with properties.

The job of the mapper is to return a Java `Object` called a `PartitionPlan` when the `mapPartitions`() method is invoked. You need to create an implementation of this interface and initialize an instance of it to return from `mapPartitions`. There is a sample implementation called `PartitionPlanImpl` that is included in the JSR-352 Reference Implementation.

To properly set up the partition plan you need to tell it all the things we provided in our JSL partition plan: number of partitions, number of threads, and the partition properties. The properties are provided as an array of `Properties` objects.

One thing to be careful with is the type of the data you put in the Properties object. A 'pure' usage of the Properties interface uses `setProperty` to put a value into the object. That method only allows you to put `Strings` in as property names and values. But you can cheat a little bit and use the underlying `HashMap put` method to put anything into the object. That works as long as you are using the matching `HashMap get` method to get them out again. But JSR-352 knows those property names and values are supposed to be `Strings` because it only uses the `Properties` object `getProperty` method to retrieve values. That will return a null if you've set non-`String` values as `Properties` and your partition won't get the value you wanted.

Partition Collector and Analyzer

In our examples so far each of our partitions has been responsible for actually doing something. But suppose you have an application where you need to merge the results of the partitions together in order to complete the work for the step. For example, suppose you need to examine every record in some giant table looking for some complicated set of things that you can't just do with a carefully crafted `SQL COUNT(*) WHERE` statement. I don't know…imagine something. Then in our basic non-partitioned job we would have a chunk step where the reader reads a record and the processor determines if this record meets our criteria. If it does, then we increment a count. Our writer doesn't do anything because we know our processor will never return anything to write. It just keeps a count. Maybe the Exit Status for the job is the count.

Before we get into how we might partition this processing…how does the Exit Status get set to the final count? Unlike the reader and writer the processor has no `close( )` method so it does not get notified when the loop is done. The simplest thing would probably be to have a step listener that lifts the value from an object shared with the processor and sets the step Exit Status as the step ends.

Ok, so now we make our step into a partitioned step and set up a plan so each partition knows what range of data to process. As each partition runs each processor object is managing its own independent count. To get the final result and Exit Status for the step we need to add up the counts from every processor instance. How do we manage this communication?

Well, you might think about the Step Context and the user data that is accessible from it. This is all part of one step, so shouldn't there be one Step Context and thus one shared user data where we could keep track of the results of each partition? Nope. It turns out that every partition gets its own separate instance of the Step Context and therefore of the user data. This is actually a good thing. It means that each partition has its own persistent user data area where it can keep track of things relevant to that partition. If the job restarts, being able to access this persisted user data might be important. Also, in some implementations of JSR-352 it is possible for the partitions to run not only on separate threads, but in separate JVMs. In those environments there is just no way for an object instance to be read/write shared across partitions.

We need another way for the partitions to communicate with the main thread of the step itself so that the Exit Status for the actual step can be properly set. That's where the collector and analyzer come in.

You configure a `collector` and `analyzer` as sub-elements of the `partition` and point them to implementations of the `PartitionCollector` and `PartitionAnalyzer` interfaces.

The collector has to implement one method, `collectPartitionData`(). That method will get control periodically during processing for the partition. For a batchlet it just gets control at the end of processing. For a chunk it gets control once at each checkpoint and again when the partition processing is complete. Each time it has the opportunity to return some `Serializable` object that can contain anything you like. In our situation we would only really need it to return anything on the final invocation and then we want to get the final

count from the processor (from some shared object, maybe hung off the `StepContext` user data) and return it.

Meanwhile the analyzer has to implement two methods: `analyzeCollectorData` and `analyzeStatus`. The analyzer methods run on the main step thread and thus have access to the `StepContext` for the step itself. There is only one instance of the analyzer (where each partition has its own instance of the collector). The `analyzeCollectorData` method gets control any time a collector provides data and is given whatever `Serializable` object the collector returned. In our scenario it would get control once for each partition since we only provide collector data at the end and it would just keep a running total of the counts provided by the collectors. Each time we could update the Exit Status for the step with the current total.

The `analyzeStatus` method receives control whenever a partition ends and it gets the Exit Status and Batch Status of the partition that ended. Both of these methods allow you to keep a running total of what happened. How do you know when the final partition has ended? Well, nothing from the runtime specifically tells you. You are presumed to know from the partition mapper how many partitions you wanted and thus you can tell in your analyzer how many results you have seen and therefore when you are on the last one.

Partition Reducer

We have one last piece of partition processing to consider. That's called the `PartitionReducer`. Despite the name, I find it easiest to think of this is just another very special listener. Consider the listeners we've already discussed and when they will get control in a partitioned step. The Step Listener will get control before and after the entire step, just like always. Within each partition (assuming a chunk step) the Chunk, Reader, Processor, and Writer Listeners will also get control, along with the various error handling listeners. But there is no listener that gives a few of partition processing. The collector and analyzer we just discussed are there to help flow the results of each partition into a single final result, but they don't really hook into the overall partition processing. There's where the `PartitionReducer` comes in.

The first point of control for the reducer is the `beginPartitionStep`() method which gets control before the mapper has even run. As the partitions then execute and the collectors feed data to the analyzer a transaction is wrapped around the analyzer processing. If an uncaught exception occurs during analyzer processing, that transaction, and any transactional updates made by the analyzer, are rolled back. Prior to rolling back the transaction, the `rollbackPartitionedStep()` method gets control.

On the other hand, if all the analyzer processing goes well, we will commit the transaction that wrapped its processing. Before that commit the `beforePartitionedStepCompletion`() method will get control. After the commit occurs the `afterPartitionedStepCompletion()` method will get control. Note that the 'after' method receives control whether we committed or rolled back and receives a status parameter that allows it to know how things turned out.

What use are these listener-like methods? Any opportunity to get control where you know the outcome can be handy. You might use these methods to help manage non-transactional assets that are manipulated by the analyzer processing. The analyzer might

stage its non-transactional results to some temporary area and these `PartitionReducer` methods might help you move those temporary results into wherever final results live (or not – depending how things went). Having this processing outside the actual analyzer processing helps protect it from any errors that might occur inside the analyzer.

Restarting a failed job

Bad things happen. Jobs fail. One of the most important features of JSR-352 is the ability to restart a failed job and have it pick up (more or less) where it left off. In this section we'll take a look at how all that works and what things you'll need to consider in writing your application so restart processing actually works (nothing is ever just magic).

What makes a job restartable?

Not every failed job can just be restarted. Remember that every job has a Batch Status value. The final Batch Status value of a job determines whether that job can be restarted or not. Obviously if the job completed, then you can't restart it. It worked, so you wouldn't want to restart it. In fact only two Batch Status values are allowed for a job to be restartable: STOPPED and FAILED.

Remember that a job can have a Batch Status of STOPPED either because an operations interface stopped it, or because a transition element said to stop it based on some Exit Status value. A job can be FAILED either because there was an unhandled exception or because a step transition said to fail it.

Assuming one of those things is true, there's one more attribute required to be able to restart a job. The job itself has to have been marked restartable in the `<job>` element. There is an attribute, defaulting to true, that says whether the job is restartable.

Job Instances and Job Executions

The difference between instances and executions turns up a lot in discussions about restart so we should quickly explain. When you submit a job, an instance of that job is created. Today at 2:00 I submitted a job so there is a 2:00-today instance of the job. It has a unique job instance identifier. When the job actually ran, a job execution was created. This is an execution for the 2:00-today job instance. If the execution completes successfully then we're done. But if fails or is stopped I might want to start it again. What I want is NOT to submit the same job, but to restart the 2:00-today instance. I want a new try at executing that instance of the job. A restart request is made for the job instance and a new, second execution is created for the original job instance.

In summary, submitting a job creates an instance. Running a job creates an execution. Restarting a failed/stopped job creates a new execution for that instance.

Restart Processing

What happens when a job is restarted? First there is the matter of job parameters. When the job was originally submitted there might have been parameters supplied as part of the submit request. When a job is restarted new parameters are supplied.

As the new job execution begins, processing begins with the first step in the JSL file, just like it did for the original execution. Probably. When a transition element in the prior execution indicated that the execution should stop, it might have also specified the restart attribute and indicated the ID of the step to begin at if the job is restarted. In that case, restart processing starts with the indicated step. Remember also that the first thing in the JSL, or the thing indicated as the restart point might not actually be a step, it could also be a flow, or a split containing flows. For simplicity, let's assume it is a normal step.

What happens next depends on the setting of the `allow-start-if-complete` and `start-limit` attributes on the step. By default `allow-start-if-complete` is false and `start-limit` is zero, meaning no limit. Let's start with that.

With the default settings job processing will examine the Batch Status for the step from the previous execution. If the previous execution resulted in a COMPLETED Batch Status for this step then it ran successfully (as far as the batch runtime is concerned) and the Exit Status from the prior execution is examined and any transition elements for the step are consulted to determine where to go next. This will be the same sequence of events that happened in the prior execution. Essentially we examine the step and branch just as we branched previously, without re-running the step. We're on a path to picking up where we left off.

But it doesn't have to be that way. The step might specify `allow-start-if-complete` to be true. In that case the JSL is indicating that we don't care what happened in the prior execution in this step. We want to run the step again and make a new decision. The step will be executed and, if it completes, the resulting Exit Status will be evaluated with any provided transition elements to determine where to go from here. That might be the same thing that happened in the prior execution. This step worked previously, so it might work again, perhaps we just needed to re-run it for some reason. Maybe we proceed to the same next step as we did before, but maybe we head off in a new direction in the JSL.

There is a limit to how many times we will re-run a step. If the `start-limit` is specified along with the `allow-start-if-complete` attribute set to true, we will only re-run the step on restarts the specified number of times. After the specified number of attempts the job will fail when execution reaches the step.

Processing for a flow or for a split containing multiple flows proceeds in just the same way, evaluating results from the prior execution again or re-executing a step when told.

And thus we proceed, branching through the JSL using results from the previous execution for completed steps, unless we re-execute a step. If we re-execute a step we might get different results and maybe branch off into new territory in the JSL with steps we have never run before. Those steps just run as normal and hopefully the job works this time.

But we might get to a step that we have run before but whose Batch Status is not COMPLETED. This is probably the step that failed or where we were when the job stopped (by command or by transition element). Now we have to "pick up where we left off" in this step. What does that mean?

Restarting a Batchlet

This is pretty easy. On the previous execution of this job instance this step failed or was stopped. We restart the step so we just run it again. The batchlet is given control and does whatever it does. I thought we should have something easy before the fun starts..

Restarting a Chunk

A chunk step has been reading and processing and writing results of the processing and committing transactions around all of that. Part of the commit processing has committed updates to information in the Job Repository based on the `Serializable Object` returned by calls to the `ItemReader` and `ItemWriter checkPointInfo()` methods.

On a restart of a chunk step the `open()` methods are called for the reader and writer and the last provided instance of `checkPointInfo` is provided. It is up to the implementation of the `open`() method to understand the checkpoint information it provided and determine how to use that to pick up where the step left off previously. This is the part where we wave our hands and call it magic. The application code, when providing the checkpoint information, has to also have code in `open()` to interpret that information and use it appropriately. That might mean advancing a cursor to the right record in a query or it might just mean opening a file for append instead of creating a new file. It depends what the application is doing.

After that it is all processing as normal for the chunk step. Unless it was partitioned. Then it gets really fun.

Restarting a Partitioned Step

If you code the partition plan directly in the JSL then this is easy. Each partition will be defined exactly as on the previous execution. Partitions that completed will not be executed again. Partitions that failed will be restarted appropriately (depending on whether it was a batchlet or chunk step). Partitions that never got a chance to run are still eligible to run this time, from the start.

But what if you have a `partitionMapper`? That allows you to dynamically create a partition plan. Does it have to create the same plan as it did for the prior execution? It can if it wants to.. The key is whatever the new partition plan returns for the `partitionsOverride` value. If set to true then the mapper is deciding that it doesn't want to remember anything from a previous execution. All new properties are being established to run brand new partitions from scratch. No checkpoint data or anything else is preserved from before.

On the other hand, if `partitionsOverride` is false then the mapper is saying that it wants to restart using the checkpoint information from before. For that to work, the mapper has to define a plan that has the same number of partitions as the plan for the previous execution. That will allow the batch runtime to use the checkpoint information from the previous partitions for a matching partition in the new execution. The mapper does have

the option to provide different properties to each partition. This might include some clue that the partition is being restarted or other guidance on how to behave.

Restarting a Decider

The last thing we'll consider is restarting a decider step. Unlike a normal step which may or may not be run on a restart, if we reach a decider it is always run. The decider is normally passed the Exit Status values from the flow(s) before it. That still happens. In the case of a restart some of the Exit Status values might be carried over from the previous execution if the step that set it wasn't run as part of this execution. Other Exit Status values might be new values if they come from steps that were run for this execution. Whatever values are provided, the decider makes a new decision and that resulting Exit Status is used with the transition elements in the decider to determine where job flow control goes next.

Conclusion

Hopefully this has been helpful in trying to understand at least some of the details of JSR-352. The actual specification is, of course, the final word in how things are supposed to work and if you find any contradictions between what is written here and what is in the specification you should obviously go with what you find in there (although drop me a note and let me know I got something wrong and I'll try to fix it up). Thanks!

Document change history

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

April 25, 2017	Initial Version
April 27, 2017	Add document number, minor update to PartitionMapper section
March 20, 2018	Corrected syntax for default parameters

End of WP102706