JSR 352
Batch Applications for the Java Platform

Specification Draft Edition 1.0

Chris Vignola

5/7/2012

1 Table of Contents

1 Table of Contents 2
2 Introduction to JSR 352 9
3 Domain Language of Batch 10
3.1 Job 11
3.1.1 JobInstance 12
3.1.2 JobParameters 12
3.1.3 JobExecution 13
3.2 Step 13
3.2.1 StepExecution 14
3.3 JobOperator 14
3.4 Item Reader 14
3.5 Item Writer 15
3.6 Item Processor 15
3.7 Chunk-oriented Processing 15
3.8 Batch Checkpoints 16
4 Job Specification Language 16
4.1 Job 16
4.1.1 Job Level Listeners 17
4.1.2 Job Level Properties 17
4.2 Step 18
4.2.1 Chunk 18
4.2.1.1 Chunk Properties 20
4.2.1.2 Chunk Exception Handling 21

4.2.1.2.1 Skipping Exceptions 21
4.2.1.2.2 Retrying Exceptions 21
4.2.1.3 Controlling Rollback During Retry 22
4.2.1.4 Checkpoint Algorithm 23
4.2.1.4.1 Checkpoint Algorithm Properties 24
4.2.2 Batchlet 24
4.2.2.1 Batchlet Properties 24
4.2.3 Step Level Properties 25
4.2.4 Step Level Listeners 25
4.2.4.1 Step Level Listener Properties 26
4.2.5 Step Sequence 27
4.2.6 Step Partitioning 27
4.2.6.1 Partition Properties 28
4.2.6.2 Partition Algorithm 29
4.2.6.2.1 Algorithm Properties 30
4.2.6.3 Logical Transaction 30
4.2.6.3.1 LogicalTX Properties 31
4.2.6.4 SubJob Collector 31
4.2.6.4.1 SubJob Collector Properties 32
4.2.6.5 SubJob Analyzer 32
4.2.6.5.1 SubJob Analyzer Properties 33
4.3 Flow 33
4.3.1 Flow Partitioning 34
4.3.1.1 Partition Properties 34
4.3.1.2 Partition Algorithm 36

4.3.1.2.1 Algorithm Properties 36
4.3.1.3 Logical Transaction 37
4.3.1.3.1 LogicalTX Properties 37
4.3.1.4 SubJob Collector 38
4.3.1.4.1 SubJob Collector Properties 38
4.3.1.5 SubJob Analyzer 39
4.3.1.5.1 SubJob Analyzer Properties 39
4.4 Split 40
4.4.1 Split Concurrency 40
4.4.1.1 Logical Transaction 41
4.4.1.1.1 LogicalTX Properties 41
4.4.1.2 SubJob Collector 42
4.4.1.2.1 SubJob Collector Properties 42
4.4.1.3 SubJob Analyzer 43
4.4.1.3.1 SubJob Analyzer Properties 44
4.5 Batch and Exit Status 44
4.5.1 Batch and Exit Status for Steps 45
4.5.1.1 Step termination may also be configured in the Job XML using the fail, end, and stop elements. See the following sections for more about those element types.Fail Element 45
4.5.1.2 End Element 46
4.5.1.3 Stop Element 47
4.5.2 Batch and Exit Status for Flows 47
4.5.3 Batch and Exit Status for Splits 48
4.6 Decision 48
4.6.1 Fail Element 49
4.6.2 End Element 49

4.6.3 Stop Element 50

4.6.4 Next Element 51

4.6.5 Decision Properties 51

4.7 Property Substitution and Overrides 52

4.7.1 Property Overrides 52

4.7.1.1 Override from Job Parameters Example 52

4.7.1.2 Override from JVM Properties Example 53

4.7.1.3 Override from Job XML Example 53

4.7.1.4 Override with Inheritance Example 54

4.7.1.5 Override with Partitions Example 54

4.7.2 Property Substitution 55

4.7.2.1 Substitution from Job Parameters Example 55

4.7.2.2 Substitution from System Properties Example 56

4.7.2.3 Substitution from Job XML Example 56

4.7.2.4 Substitution with Inheritance Example 57

4.7.2.5 Substitution with Partitions Example 57

4.8 Job XML Inheritance 58

4.8.1 Merging Lists 60

4.8.2 Inheritance Namespace 61

5 Batch Programming Model 61

5.1 Steps 61

5.1.1 Chunk 61

5.1.1.1 @ItemReader 61

5.1.1.1.1 @Open 62

5.1.1.1.2 @Close 62

5.1.1.1.3 @ReadItem 63
5.1.1.1.4 @GetCheckpointInfo 63
5.1.1.2 @ItemProcessor 64
5.1.1.2.1 @ProcessItem 64
5.1.1.3 @ItemWriter 65
5.1.1.3.1 @Open 65
5.1.1.3.2 @Close 66
5.1.1.3.3 @WriteItems 66
5.1.1.3.4 @GetCheckpointInfo 67
5.1.1.4 @CheckpointAlgorithm 67
5.1.1.4.1 @GetCheckpointTimeout 67
5.1.1.4.2 @BeginCheckpoint 68
5.1.1.4.3 @IsReadyToCheckpoint 69
5.1.1.4.4 @EndCheckpoint 69
5.1.2 @Batchlet 70
5.1.2.1 @BeginStep 70
5.1.2.2 @Process 71
5.1.2.3 @Cancel 71
5.1.2.4 @EndStep 72
5.2 Listeners 72
5.2.1 @JobListener 72
5.2.1.1 @BeforeJob 73
5.2.1.2 @AfterJob 73
5.2.1.3 @BeforeStep 74
5.2.1.4 @AfterStep 74

5.2.2 @StepListener 75
5.2.2.1 @BeforeStep 75
5.2.2.2 @AfterStep 76
5.2.3 @ CheckpointListener 76
5.2.3.1 @BeforeCheckpoint 77
5.2.3.2 @AfterCheckpoint 77
5.2.4 @ItemReadListener 78
5.2.4.1 @BeforeRead 78
5.2.4.2 @AfterRead 79
5.2.4.3 @OnReadError 79
5.2.5 @ItemProcessListener 80
5.2.5.1 @BeforeProcess 80
5.2.5.2 @AfterProcess 81
5.2.5.3 @OnProcessError 81
5.2.6 @ItemWriteListener 82
5.2.6.1 @BeforeWrite 82
5.2.6.2 @AfterWrite 83
5.2.6.3 @OnWriteError 83
5.2.7 @SkipListener 84
5.2.7.1 @OnSkipInRead 84
5.2.7.2 @OnSkipInProcess 85
5.2.7.3 @OnSkipInWrite 85
5.2.8 @RetryListener 86
5.2.8.1 @OnRetryException 86
5.2.8.2 @OnRetryItem 87

5.3 Batch Properties 88
5.3.1 @BatchProperty 88
5.4 Batch Contexts 89
5.4.1 @BatchContext 89
5.4.1.1 Batch Context Lifecycle and Scope 90
5.4.1.2 Batch Context and @Decider 91
5.5 Parallelization 91
5.5.1 @PartitionAlgorithm 92
5.5.1.1 @CalculatePartitions 92
5.5.2 @LogicalTX 93
5.5.2.1 @LogicalTXBegin 93
5.5.2.2 @LogicalTXBeforeCompletion 94
5.5.2.3 @LogicalTXRollback 94
5.5.2.4 @LogicalTXAfterCompletion 95
5.5.3 @SubJobCollector 96
5.5.3.1 @CollectSubJobData 96
5.5.4 @SubJobAnalyzer 97
5.5.4.1 @AnalyzeCollectorData 98
5.5.4.2 @AnalyzeExitStatus 98
5.6 @Decider 99
5.6.1 @Decide 99
5.7 Batch Artifact Loader 100
5.7.1 Batch Artifact Loader API 100
5.7.2 batch.xml 101
6 Batch Runtime Specification 101

6.1 Job Identifiers 101

6.2 JobOperator 102

6.3 Classloader Scope 102

6.4 Job Path 102

6.5 Packaging Model 103

6.6 Java EE Environment 103

6.6.1 Transaction Handling 103

6.6.2 Batch Artifact Loading 104

6.6.3 JNDI 104

6.7 Java SE Environment 104

6.7.1 Factory Method 104

6.7.2 Transaction Handling 104

6.7.3 Batch Artifact Loading 104

6.8 Supporting Classes 105

6.8.1 JobContext 105

6.8.2 StepContext 106

6.8.3 FlowContext 108

6.8.4 SplitContext 109

6.8.5 PartitionPlan 109

6.8.6 JobOperator 110

7 Credits 114

2 Introduction to JSR 352

Batch processing is a pervasive workload pattern, expressed by a distinct application organization and execution model. It is found across virtually every industry, applied to such tasks as statement

generation, bank postings, risk evaluation, credit score calculation, inventory management, portfolio optimization, and on and on. Nearly any bulk processing task from any business sector is a candidate for batch processing.

Batch processing is typified by bulk-oriented, non-interactive, background execution. Frequently long-running, it may be data or computationally intensive, execute sequentially or parallel, and may be initiated through various invocation models, including ad hoc, scheduled, and on-demand.

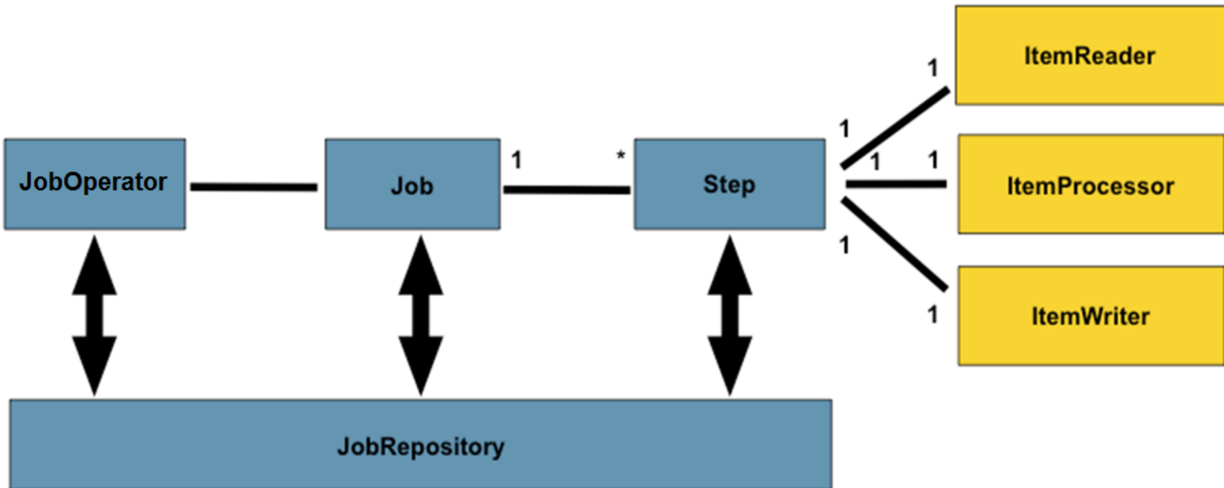
Batch applications have common requirements, including logging, checkpoint, and parallelization. Batch workloads have common requirements, especially operational control, which allow for initiation of, and interaction with, batch instances; such interactions include stop and restart.

3 Domain Language of Batch

To any experienced batch architect, the overall concepts of batch processing used in by JSR 352 should be familiar and comfortable. There are "Jobs" and "Steps" and developer supplied processing units called ItemReaders and ItemWriters. However, because of the JSR 352 operations, callbacks, and idioms, there are opportunities for the following:

- significant improvement in adherence to a clear separation of concerns
- clearly delineated architectural layers and services provided as interfaces
- simple and default implementations that allow for quick adoption and ease of use out-of-the-box
- significantly enhanced extensibility

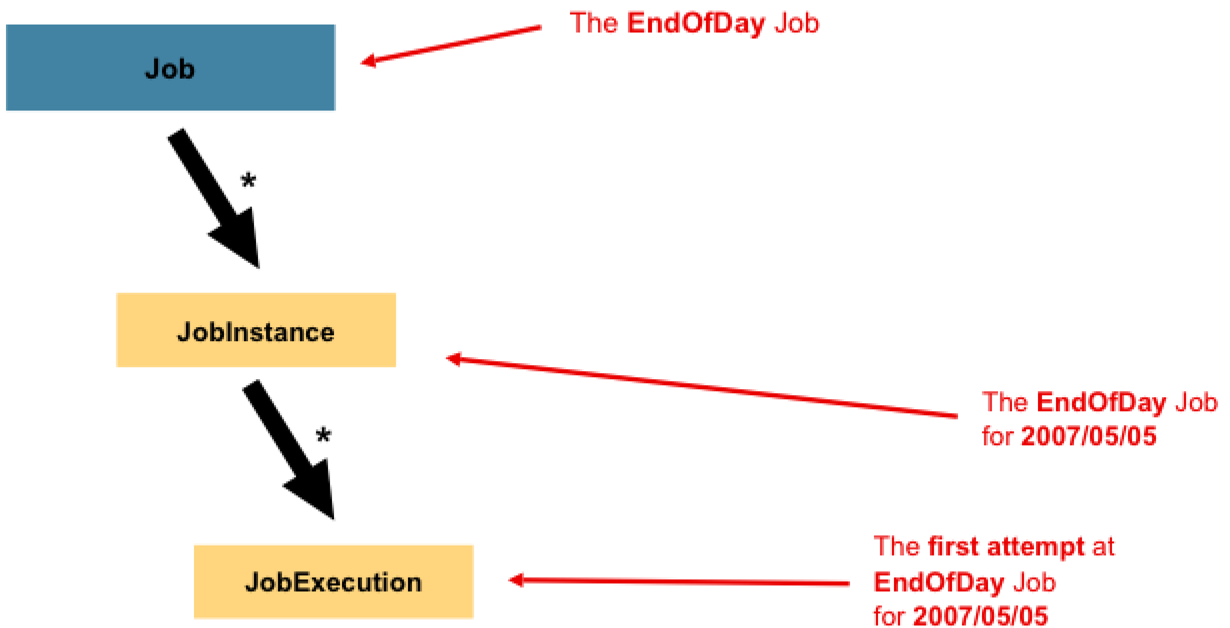
The diagram below is simplified version of the batch reference architecture that has been used for decades. It provides an overview of the components that make up the domain language of batch processing. This architecture framework is a blueprint that has been proven through decades of implementations on the last several generations of platforms (COBOL/Mainframe, C++/Unix, and now Java/anywhere). JCL and COBOL developers are likely to be as comfortable with the concepts as C++, C# and Java developers. JSR 352 specifies the layers, components and technical services commonly found in robust, maintainable systems used to address the creation of simple to complex batch applications.



The diagram above highlights the key concepts that make up the domain language of batch. A Job has one to many steps, which has exactly one ItemReader, ItemProcessor, and ItemWriter. A job needs to be launched (JobOperator), and meta data about the currently running process needs to be stored (JobRepository).

3.1 Job

A Job is an entity that encapsulates an entire batch process. A Job will be wired together via a Job XML file. However, Job is just the top of an overall hierarchy:



With JSR 352, a Job is simply a container for Steps. It combines multiple steps that belong logically together in a flow and allows for configuration of properties global to all steps, such as restartability. The job configuration contains:

- The simple name of the job
- Definition and ordering of Steps
- Whether or not the job is restartable

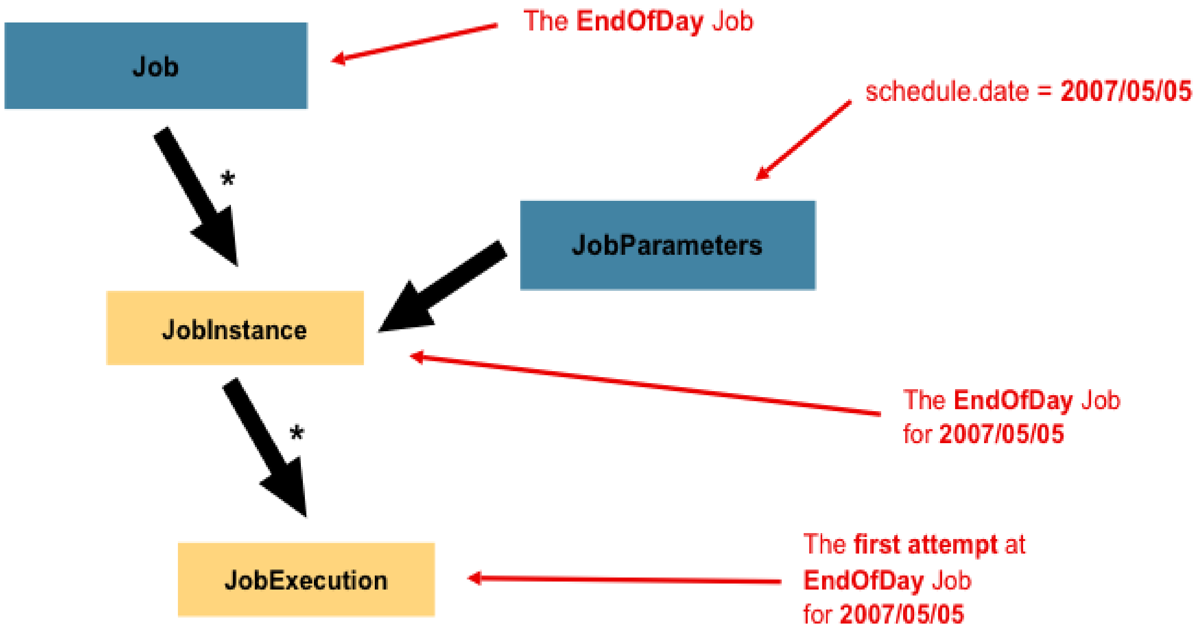
3.1.1 JobInstance

A JobInstance refers to the concept of a logical job run. Let's consider a batch job that should be run once at the end of the day, such as the 'EndOfDay' job from the diagram above. There is one 'EndOfDay' Job, but each individual run of the Job must be tracked separately. In the case of this job, there will be one logical JobInstance per day. For example, there will be a January 1st run, and a January 2nd run. If the January 1st run fails the first time and is run again the next day, it is still the January 1st run. (Usually this corresponds with the data it is processing as well, meaning the January 1st run processes data for January 1st, etc). Therefore, each JobInstance can have multiple executions (JobExecution is discussed in more detail below); one or many JobInstances corresponding to a particular Job and JobParameters can be running at a given time.

The definition of a JobInstance has absolutely no bearing on the data that will be loaded. It is entirely up to the ItemReader implementation used to determine how data will be loaded. For example, in the EndOfDay scenario, there may be a column on the data that indicates the 'effective date' or 'schedule date' to which the data belongs. So, the January 1st run would only load data from the 1st, and the January 2nd run would only use data from the 2nd. Because this determination will likely be a business decision, it is left up to the ItemReader to decide. What using the same JobInstance will determine, however, is whether or not the 'state' (i.e. the JobContext, which is discussed below) from previous executions will be used. Using a new JobInstance will mean 'start from the beginning' and using an existing instance will generally mean 'start from where you left off'.

3.1.2 JobParameters

Having discussed JobInstance and how it differs from Job, the natural question to ask is: "how is one JobInstance distinguished from another?" The answer is: JobParameters. JobParameters is a set of parameters used to start a batch job. They can be used for identification or even as reference data during the run:



In the example above, where there are two instances, one for January 1st, and another for January 2nd, there is really only one Job, one that was started with a job parameter of 01-01-2008 and another that was started with a parameter of 01-02-2008. Thus, the contract can be defined as: JobInstance = Job + JobParameters. This allows a developer to effectively control how a JobInstance is defined, since they control what parameters are passed in.

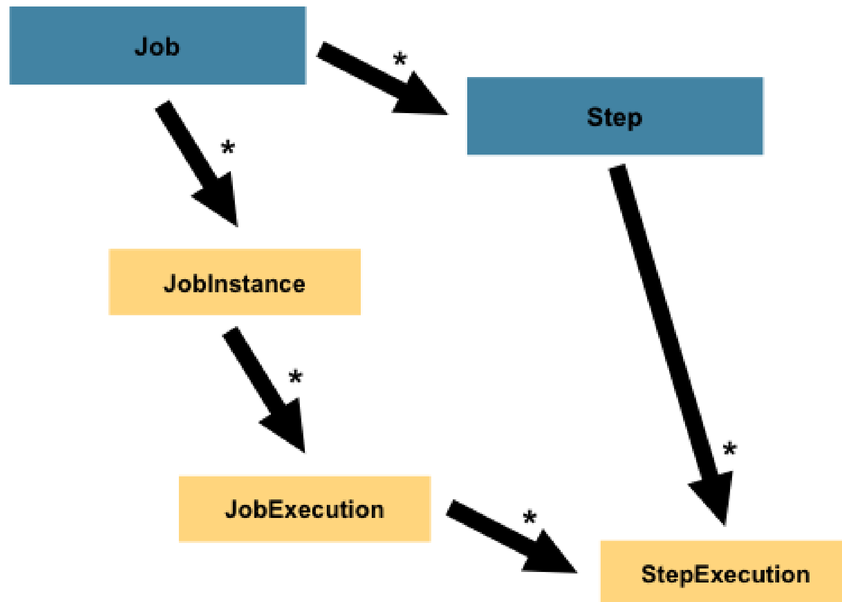
3.1.3 JobExecution

A JobExecution refers to the technical concept of a single attempt to run a Job. An execution may end in failure or success, but the JobInstance corresponding to a given execution will not be considered complete unless the execution completes successfully. Using the EndOfDay Job described above as an example, consider a JobInstance for 01-01-2008 that failed the first time it was run. If it is run again with the same job parameters as the first run (01-01-2008), a new JobExecution will be created. However, there will still be only one JobInstance.

3.2 Step

A Step is a domain object that encapsulates an independent, sequential phase of a batch job. Therefore, every Job is composed entirely of one or more steps. A Step contains all of the information necessary to define and control the actual batch processing. This is a necessarily vague description because the contents of any given Step are at the discretion of the developer writing a Job. A Step can be as simple or complex as the developer desires. A simple Step might load data from a file into the database, requiring little or no code. (depending upon the implementations used) A more complex Step may have

complicated business rules that are applied as part of the processing. As with Job, a Step has an individual StepExecution that corresponds with a unique JobExecution:



3.2.1 StepExecution

A StepExecution represents a single attempt to execute a Step. A new StepExecution will be created each time a Step is run, similar to JobExecution. However, if a step fails to execute because the step before it fails, there will be no execution persisted for it. A StepExecution will only be created when its Step is actually started.

Step executions are represented by objects of the StepExecution class. Each execution contains a reference to its corresponding step and JobExecution, and transaction related data such as commit and rollback count and start and end times. Additionally, each step execution has a corresponding StepContext, which contains any data a developer needs persisted across batch runs, such as statistics or state information needed to restart.

3.3 JobOperator

JobOperator provides an interface to manage all aspects of job processing, including operational commands, such as start, restart, and stop, as well as job repository related commands, such as retrieval of job and step executions. See section [REF] for more details about JobOperator.

3.4 Item Reader

ItemReader is an abstraction that represents the retrieval of input for a Step, one item at a time. When the ItemReader has exhausted the items it can provide, it will indicate this by returning null. See section

[REF] for more details about ItemReaders.

3.5 Item Writer

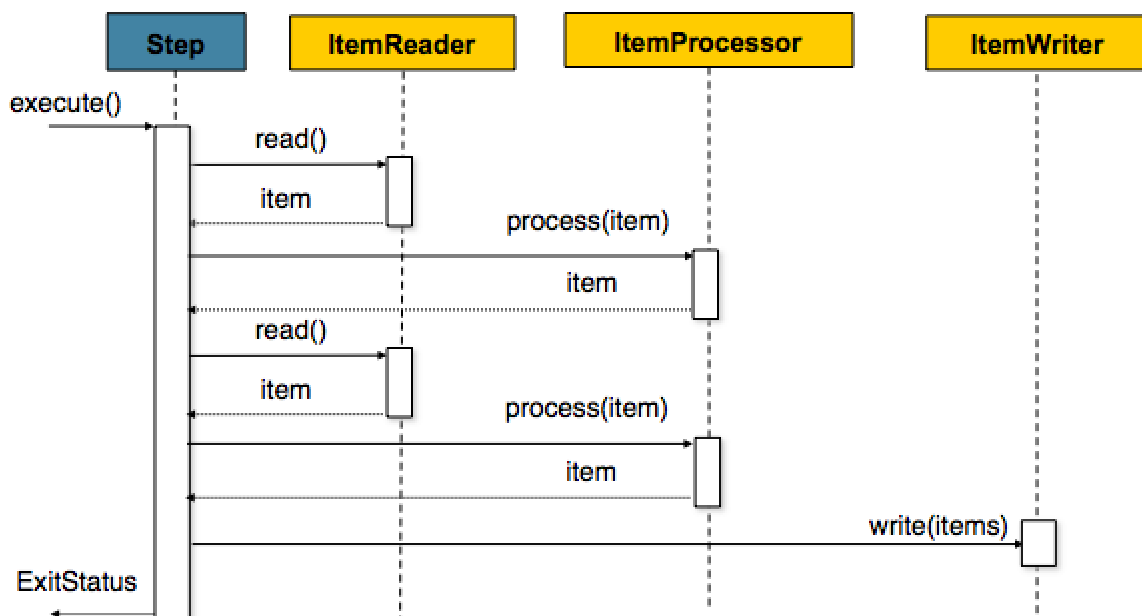
ItemWriter is an abstraction that represents the output of a Step, one batch or chunk of items at a time. Generally, an item writer has no knowledge of the input it will receive next, only the item that was passed in its current invocation. See section [REF] for more details about ItemWriters.

3.6 Item Processor

ItemProcessor is an abstraction that represents the business processing of an item. While the ItemReader reads one item, and the ItemWriter writes them, the ItemProcessor provides access to transform or apply other business processing. If, while processing the item, it is determined that the item is not valid, returning null indicates that the item should not be written out. See section [REF] for more details about ItemProcessors.

3.7 Chunk-oriented Processing

JSR 352 specifies a 'Chunk Oriented' processing style as its primary pattern. Chunk oriented processing refers to reading the data one at a time, and creating 'chunks' that will be written out, within a transaction boundary. One item is read in from an ItemReader, handed to an ItemWriter, and aggregated. Once the number of items read equals the commit interval, the entire chunk is written out via the ItemWriter, and then the transaction is committed.



3.8 Batch Checkpoints

For data intensive batch applications - particularly those that may run for long periods of time - checkpoint/restart is a common design requirement. Checkpoints allow a step execution to periodically bookmark its current progress to enable restart from the last point of consistency, following a planned or unplanned interruption.

Checkpoints work naturally with chunk-oriented processing. The end of processing for each chunk is a natural point for taking a checkpoint.

JSR 352 specifies runtime support for checkpoint/restart in a generic way that can be exploited by any batch step that has this requirement.

Since progress during a step execution is really a function of the current position of the input/output data, natural placement of function suggests the knowledge for saving/restoring current position is a reader/writer responsibility.

Since managing step execution is a container responsibility, the batch runtime must necessarily understand step execution lifecycle, including initial start, execution end states, and restart.

Since checkpoint frequency has a direct effect on lock hold times, for lockable resources, tuning checkpoint interval size can have a direct bearing on overall system throughput. Ideally, this should be adjustable operationally and not explicitly controlled by the batch application itself.

4 Job Specification Language

Job Specification Language (JSL) specifies a job, its steps, and directs their execution. The JSL for JSR 352 is implemented with XML and will be henceforth referred to as "Job XML".

4.1 Job

The 'job' element identifies a job.

Syntax:

<job id="{*name*}" restartable="{*true*|*false*}" abstract ="{*true*|*false*}" parent="{*name*}">

Where:

id	Specifies the logical *name* of the job and is used for identification purposes. It must be a valid XML string value. This is a required attribute.
restartable	Specifies whether or not this job is restartable . See section [REF] for specification of

	restart processing. It must specify a valid XML boolean value. This is an optional attribute. The default is *true.*
abstract	Specifies whether or not this job is an abstract job. Abstract jobs are used for job inheritance through the parent attribute. The elements of an abstract job are combined with the elements of the inheriting job. It must specify a valid XML boolean value. This is an optional attribute. The default is *false.*
parent	Specifies the id value of a job definition to be incorporated into the current job. It must be a valid XML string value. This is an optional attribute. The default is no parent.

4.1.1 Job Level Listeners

Job level listeners may be configured to a job in order to intercept job execution. The listener element may be specified as sub-element of the job element for this purpose. Job listener is the only listener type that may be specified as a job level listener.

Multiple listeners may be configured on a job. A job listener is invoked according to its relationship to the job life cycle. See section [REF] to understand the sequence of invocation of the batch artifacts in a job's life cycle. If multiple job listeners are configured on the same job, they are invoked in the order they are specified.

Syntax:

<listener ref="{*name*}">

Where:

ref	Specifies the name of a batch artifact, as known to the batch artifact loader. See section [REF] for further information about batch artifact names and batch artifact loading.

4.1.2 Job Level Properties

The 'properties' element may be specified as a sub-element of the job element. It is used to expose properties to any batch artifact belonging to the job and also to the batch runtime. Any number of properties may be specified. See section [REF] for further information on how job level properties may be used by the batch runtime. Step level properties are available through the Job Context runtime object. See section [REF] for further information about Job Context.

Syntax:

<properties> <property name="{*property-name*}" value="{ *name-value*}" />
</properties>

Where:

name	Specifies a unique property name. It must be a valid XML string value. This is a required attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See section [REF] for further information on substitution symbols.

4.2 Step

The 'step' element identifies a job step and it's characteristics. Step is a sub-element of job. A job may contain any number of steps.

Syntax:

<step id="*{name}*" start-limit="*{integer}*"
allow-start-if-complete ="*{true|<u>false</u>}*" next="*{ flow-id|step-id|split-id|decision-id}*"
abstract="*{<u>true</u>|false}*" parent="*{ name}*">

Where:

id	Specifies the logical *name* of the step and is used for identification purposes. It must be a valid XML string value. This is a required attribute.
start-limit	Specifies the number of times this step may be restarted for a restartable job. It must be a valid XML integer value. This is an optional attribute. The default is 0, which means no limit. See section [REF] for specification of restart processing.
allow-start-if-complete	Specifies whether this step is allowed to start during job restart, even if the step completed in a previous execution. It must be a valid XML boolean value. A value of true means the step is allowed to restart. This is an optional attribute. The default is false.
next	Specifies the next step, flow, split, or decision to run after this step is complete. It must be a valid XML string value. This is an optional attribute. The default is false
abstract	Specifies whether or not this step is an abstract step. Abstract step are used for step inheritance through the parent attribute. The elements of an abstract step are combined with the elements of the inheriting step.
parent	Specifies the id value of a step definition to be used by the current step.

4.2.1 Chunk

The 'chunk' element identifies a chunk type step. It is a sub-element of the step element. A chunk is a type of step. A chunk implements the reader-processor-writer pattern of batch. A chunk runs in the scope of a transaction. A chunk may be configured so the batch runtime periodically checkpoints the progress of the step by committing the current transaction and starting a new transaction scope. A

chunk that is not complete is restartable from its last checkpoint.   A chunk that is complete and belongs to a step configured with allow-start-if-complete=true runs from the beginning when restarted.

Syntax:

```
<chunk reader="{ref}"
        processor="{ ref}"
        writer="{ ref}"
        checkpoint-policy="{record|time|custom}"
        commit-interval="{value}"
        buffer-reads="{true|false}"
        chunk-size="{value}"
        skip-limit="{value}"
        retry-limit="{value}"
/>
```

Where:

reader	Specifies the batch artifact *name* of an item reader. It must be a valid XML string value. It is a required attribute.
processor	Specifies the batch artifact *name* of an item processor. It must be a valid XML string value. It is a required attribute.
writer	Specifies the batch artifact *name* of an item writer. It must be a valid XML string value. It is a required attribute.
checkpoint-policy	Specifies the checkpoint policy that governs commit behavior for this chunk. Valid values are: "item", "time", or "custom". The "item" policy means the chunk is checkpointed after a specified number of items are processed. The "time" policy means the chunk is checkpointed after a specified amount of time. The "custom" policy means the chunk is checkpointed according to a checkpoint algorithm implementation. Specifying "custom" requires that the checkpoint-algorithm element is also specified. There is also a built-in "item-time" custom policy - see checkpoint-algorithm element for further information. It is an optional attribute. The default policy is "item".
commit-interval	Specifies the commit interval for the specified checkpoint policy. It must be valid XML integer. It is an optional attribute. The default is 10. The unit meaning of the commit-interval specifies depends on the specified checkpoint policy. For "item" policy, commit-interval specifies a number of items. For "time" policy, commit-interval specifies a number of seconds. The commit-interval attribute is ignored for "custom" policy.
buffer-reads	Specifies whether the step buffers the item it reads before writing them. It must be a valid XML boolean value. A value of 'true' means reads are buffered. A common reason for specifying 'false' is when the reader is transactional and will be reset if the current transaction is rolled back - e.g. a JMS queue reader. It is an optional attribute. The default is true.
chunk-size	Specifies the number of items buffered when a step is buffering reads. It must be a valid XML integer value. It is an optional attribute. When buffer-reads is true,

	the default is the value of the commit-interval attribute, when checkpoint-policy is 'item', else the default is 10. If buffer-reads is false, chunk-size is always 1, no matter what chunk-size value is specified.
skip-limit	Specifies the number of exceptions a step will skip if any configured skippable exceptions are thrown by chunk processing. It must be a valid XML integer value. It is an optional attribute. The default is no limit.
retry-limit	Specifies the number of times a step will retry if any configured retryable exceptions are thrown by chunk processing. It must be a valid XML integer value. It is an optional attribute. The default is no limit.

4.2.1.1 Chunk Properties

The 'properties' element may be specified as a sub-element of the chunk element. It is used to pass property values to the reader, processor, and writer batch artifacts of a chunk type step. There is a distinct sub-element for each batch artifact type on a chunk type step. Any number of properties may be specified.Syntax:

< property name="{*property-name*}" value="{ *name-value*}" target="{*artifact-name*}"/>

Where:

name	Specifies a unique property name. It must be a valid XML string value. It should match a property name specified in a corresponding batch artifact. This is a required attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See section [REF] for further information on substitution symbols.
target	Specifies the target batch artifact type to which the property must be applied. Valid values are: "reader", "processor", "writer". This is an optional attribute. If not specified, the property applies to all batch artifacts in the current chunk.

Example:

The following chunk type step sends the property named "filename" to both the reader and to the writer, with each property instance having a different value:

```
<step id="Chunk1">
        <chunk reader="MyReader" processor="MyProcessor" writer="MyWriter">
                <properties>
                        <property name="filename" value="/tmp/infile.txt" target=reader">
                        <property name="filename" value="/tmp/outfile.txt" target="writer">
                </properties>
```

```
	</chunk>
</step>
```

4.2.1.2 Chunk Exception Handling

4.2.1.2.1 Skipping Exceptions

The skippable-exception-classes element specifies a set of exceptions that chunk processing should skip. This element is a sub-element of the chunk element. It applies to exceptions thrown from the reader, processor, and writer batch artifacts of a chunk type step. The total number of skips is set by the skip-limit attribute on the chunk element. See section [REF] for details on the chunk element.

The optional Skip Listener batch artifact can be configured to the step or job. A Skip Listener can receive control for a skip in the reader, processor, or writer. See section [REF] for details on the Skip Listener batch artifact.

Syntax:

```
<skippable-exception-classes>
	<include class="{class name}"/>
	<exclude class="{class name}"/>
</skippable-exception-classes>
```

Where:

include class	Specifies the class name of an exception to skip. It must be a valid XML string value. The include sub-element is optional. However, when specified, the class attribute is required.
exclude class	Specifies a class name of an exception to not skip. It must be a valid XML string value. The exclude sub-element is optional. However, when specified, the class attribute is required.

4.2.1.2.2 Retrying Exceptions

The retryable-exception-classes element specifies a set of exceptions that chunk processing should retry. This element is a sub-element of the chunk element. It applies to exceptions thrown from the writer batch artifacts of a chunk type step. The total number of retry attempts is set by the retry-limit attribute on the chunk element. See section [REF] for details on the chunk element.

The optional Retry Listener batch artifact can be configured to the step or job. A Retry Listener receives control for retryable exceptions thrown by a writer. See section [REF] for details on the Retry Listener batch artifact.

When a retryable exception occurs, the default behavior is for the batch runtime to rollback the current transaction and re-process the current chunk. Chunk re-processing follows this sequence of execution:

1. a new transaction scope started;
2. the items in the current chunk up to, but not including, this item are re-processed by the item writer;
3. the current item is processed through the @OnRetryItem method of all configured Retry Listeners, if any;
4. the current item is then processed through the item writer;
5. the remaining items in the current chunk are processed through the item writer.

The default retry behavior can be overridden by configuring the no-rollback-exception-classes element. See section [REF] for more information on specifying no-rollback exceptions. If a retryable exception is also specified as a no-rollback exeception, the batch runtime does not rollback the current exception, rather, it simply re-processes the current item through the item writer. Retry listeners (if configured) are called before the item writer.

Syntax:

```
<retryable-exception-classes>
	<include class="{class name}"/>
	<exclude class="{class name}"/>
</retryable-exception-classes>
```

Where:

include class	Specifies a class name of an exception to retry. It must be a valid XML string value. The include sub-element is optional. However, when specified, the class attribute is required.
exclude class	Specifies a class name of an exception to not retry. It must be a valid XML string value. The exclude sub-element is optional. However, when specified, the class attribute is required.

4.2.1.3 Controlling Rollback During Retry

The no-rollback-exception-classes element specifies a list of exceptions that override the default retry behavior of rollback for retryable exceptions. This element is a sub-element of the chunk element. If a retryable exception is thrown the default behavior is to rollback and re-process the chunk. If the retryable exception is also specified as a no-rollback exception, then no rollback occurs and only the current item is re-processed. A no-rollback exception is ignored if it is not also specified as a retryable exception. See section [REF] for more details about retry processing.

Syntax:

```
<no-rollback-exception-classes>
        <include class="{class name}"/>
        <exclude class="{class name}"/>
</no-rollback-exception-classes>
```

Where:

include class	Specifies a class name of an exception for which rollback will not occur during retry processing. It must be a valid XML string value. The include sub-element is optional. However, when specified, the class attribute is required.
exclude class	Specifies a class name of an exception for which rollback will occur during retry processing. It must be a valid XML string value. The exclude sub-element is optional. However, when specified, the class attribute is required.

4.2.1.4 Checkpoint Algorithm

The checkpoint-algorithm element specifies an optional custom checkpoint algorithm. It is a sub-element of the chunk element. It is valid when the chunk element checkpoint-policy attribute specifies the value 'custom'. A custom checkpoint algorithm may be used to provide a checkpoint decision based on factors other than only number of records, or amount of time. When in use, the custom checkpoint policy @IsReadyToCheckpoint method is invoked after every invocation of the item processor and directs the batch runtime when it is time to end a checkpoint interval. See section [REF] for further information about custom checkpoint algorithms.

Syntax:

```
<checkpoint-algorithm ref="{name}"/>
```

Where:

ref	Specifies the name of a batch artifact, as known to the batch artifact loader. See section [REF] for further information about batch artifact names and batch artifact

	loading.

4.2.1.4.1 Checkpoint Algorithm Properties

The 'properties' element may be specified as a sub-element of the checkpoint algorithm element. It is used to pass property values to any of the available listener types. Any number of properties may be specified.

Syntax:

<properties> <property name="{*property-name*}" value="{ *name-value*}" />
</properties>Where:

name	Specifies a unique property name. It must be a valid XML string value. It should match a property name specified in a corresponding batch artifact. This is a required attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See section [REF] for further information on substitution symbols.

4.2.2 Batchlet

The batchlet element specifies a task-oriented batch step. It is specified as a sub-element of the step element. It is mutually exclusive with the chunk element. The batchlet element identifies a batch artifact implemented using the @Batchlet annotation. Steps of this type are useful for performing a variety of tasks that are not item-oriented, such as executing a command or doing file transfer.

Syntax:

<batchlet ref="{*name*}"/>

Where:

ref	Specifies the name of a batch artifact, as known to the batch artifact loader. See section [REF] for further information about batch artifact names and batch artifact loading.

4.2.2.1 Batchlet Properties

The 'properties' element may be specified as a sub-element of the batchlet element. It is used to pass property values to a batchlet. Any number of properties may be specified.

Syntax:

<properties> <property name="{*property-name*}" value="{ *name-value*}" />
</properties>

Where:

name	Specifies a unique property name. It must be a valid XML string value. It should match a property name specified in a corresponding batch artifact. This is a required attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See section [REF] for further information on substitution symbols.

4.2.3 Step Level Properties

The 'properties' element may be specified as a sub-element of the step element. It is used to expose properties to any step level batch artifact and also to the batch runtime. Any number of properties may be specified. See section [REF] for further information on how step level properties may be used by the batch runtime. Step level properties are available through the Step Context runtime object. See section [REF] for further information about Step Context.

Syntax:

<properties> <property name="{*property-name*}" value="{ *name-value*}" />
</properties>

Where:

name	Specifies a unique property name. It must be a valid XML string value. This is a required attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See section [REF] for further information on substitution symbols.

4.2.4 Step Level Listeners

Step level listeners may be configured to a job step in order to intercept step execution. The listener element may be specified as sub-element of the step element for this purpose. The following listener types may be specified according to step type:

- chunk step - step listener, item read listener, item process listener, item write listener, checkpoint listener, skip listener, and retry listener
- batchlet step - step listener

Multiple listeners may be configured on a job step. The listeners are invoked according to their relationship to the job step life cycle. See section [REF] to understand the sequence of invocation of the batch artifacts in a job's life cycle. If multiple listeners of the same type (e.g. step listener) are configured on the same job step, they are invoked in the order they are specified.

Syntax:

<listener ref="{*name*}">

Where:

ref	Specifies the name of a batch artifact, as known to the batch artifact loader. See section [REF] for further information about batch artifact names and batch artifact loading.

4.2.4.1 Step Level Listener Properties

The 'properties' element may be specified as a sub-element of the step-level listeners element. It is used to pass property values to any of the available listener types. Any number of properties may be specified.

Syntax:

<properties> <property name="{*property-name*}" value="{ *name-value*}" /> </properties>Where:

name	Specifies a unique property name. It must be a valid XML string value. It should match a property name specified in a corresponding batch artifact. This is a required attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See section [REF] for further information on substitution symbols.

4.2.5 Step Sequence

The first step, flow, or split defines the first step (or steps) to execute for the a given Job XML. The 'next' attribute on the step, flow, or split defines what executes next. The next attribute may specify a step, flow, split, or decision. The next attribute is supported on step, flow, and split elements. Steps may alternatively use the "next" *element* to specify what executes next. The next attribute and next element may not be specified together in the same step.

Syntax:

<next on="{*exit status*}" to="{*id*}" />

Where:

on	Specifies an exit status to match to the current next element. It must be a valid XML string value. Wildcards of "*" and "?" may be used. "*" matches one or more characters. "?" matches exactly one character. It must match an exit status value in order to have effect. This is a required attribute.
to	Specifies the id of another step, split, flow, or decision, which will execute next. It must be a valid XML string value. It must match an id of another step, split, flow, or decision in the same job. For a step inside a flow, the id must match another step in the same flow. This is a required attribute.

Multiple next elements may be specified for a given step. When multiple next elements are specified, they are ordered from most specific to least specific. The current exit status value is then compared against each next element until a match is found. When the next element is specified, all possible exit status values must be accounted for. If an exit status occurs that does not match one of the next elements, the job ends in the failed state.

4.2.6 Step Partitioning

A batch step may run as a partitioned step. A partitioned step runs as multiple instances of the same step definition across multiple threads, one partition per thread. The number of threads is controlled through either a static specification in the Job XML or through a batch artifact called a partition algorithm. The use of a partition algorithm overrides the static specification.

Each instances needs the ability to receive unique parameters to instruct it which data on which to operate. Properties for each partition may be specified statically in the Job XML or through the optional partition algorithm.Since each thread runs a separate copy of the step, chunking and checkpointing

occur independently on each thread for chunk type steps. There needs to optionally be a way to coordinate these separate units of work in a logical transaction so that backout is possible if one or more threads experience failure. The LogicalTX batch artifact provides a way to do that. A LogicalTX provides programmatic control over logic unit of work demarcation that scopes all threads of a partitioned step.

The threads executing a partitioned step may need to share results with a control point to decide the overall outcome of the step. The collector/analyzer batch artifact provides for this need.Syntax:

<partition instances="{*number*}"/>

Where:

instances	Specifies the number of partition instances (threads) for this partitioned step. This is a an optional attribute. The default is 1.

Example:

The following Job XML snippet shows how to specify a step partitioned onto 2 threads:

```
<step id="Step1">
        <chunk .../>
        <partition instances="2"/>
</step>
```

4.2.6.1 Partition Properties

When defining a statically partitioned step, it is possible to specify unique property values to pass to each partition directly in the Job XML using the property element.

Syntax:

```
<properties>
        <property name="{property-name}.${partition-number}" value="{ name-value}" />
</properties>
```

Where:

name	Specifies a unique property name and partition number. It must be a valid XML string value. It should match a property name specified in a corresponding batch artifact. This is a required attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See section [REF] for further information on substitution symbols.

Example:

The following Job XML snippet shows a 2-way partition (i.e. 2 threads) with a unique value for the property named "filename" for each partition:

```
<partition instances="2">
	<properties>
		<property name="filename.1" value="/tmp/file1.txt"/>
		<property name="filename.2" value="/tmp/file2.txt"/>
	</properties>
</partition
```

The "target" attribute may be specified on the property element to apply that property to a specific batch element on the step if there is more than one possible target batch artifact.

E.g.

```
<step id="Chunk1">
	<chunk reader="MyReader" processor="MyProcessor" writer="MyWriter"
		<properties>
			<property name="filename.in" value=" " target="reader">
			<property name="filename.out" value=" "  target="writer">
		</properties>
	</chunk>
	<partition instances="2">
		<properties>
			<property name="filename.in.1" value="/tmp/infile1.txt" >
			<property name="filename.out.1" value="/tmp/outfile1.txt" >
			<property name="filename.in.2" value="/tmp/infile2.txt" >
			<property name="filename.out.2" value="/tmp/outfile2.txt">
		</properties>
	</partition
</step>
```

4.2.6.2 Partition Algorithm

The partition algorithm provides a programmatic means for calculating the number of partitions for a partitioned step. The partition algorithm also specifies the properties for each partition. The use of a partition algorithm overrides the 'instances' attribute on the parent partition element, and also

overrides any batch properties specified on that same element. The algorithm element specifies a reference to a PartitionAlgorithm batch artifact; see section [REF] for further information.

Syntax:

<algorithm ref="{*name*}">

Where:

ref	Specifies the name of a batch artifact, as known to the batch artifact loader. See section [REF] for further information about batch artifact names and batch artifact loading.

4.2.6.2.1 Algorithm Properties

The 'properties' element may be specified as a sub-element of the algorithm element. It is used to pass property values to a PartitionAlgorithm batch artifact. Any number of properties may be specified.

Syntax:

<properties>
<property name="{*property-name*}" value="{ *name-value*}" />
</properties>

Where:

name	Specifies a unique property name. It must be a valid XML string value. It should match a property name specified in a corresponding batch artifact. This is a required attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See section [REF] for further information on substitution symbols.

4.2.6.3 Logical Transaction

A partitioned step executes in the context of a logical transaction. A logical transaction provides a kind of unit of work demarcation around the execution of the collection of threads. Programmatic interception of the logical transaction lifecycle is possible by providing a logical transaction. The logicalTX element specifies a reference to a LogicalTX batch artifact; see section [REF] for further information.

Syntax:

<logicalTX ref="{*name*}">

Where:

ref	Specifies the name of a batch artifact, as known to the batch artifact loader. See section [REF] for further information about batch artifact names and batch artifact loading.

4.2.6.3.1 LogicalTX Properties

The 'properties' element may be specified as a sub-element of the logicalTX element. It is used to pass property values to a LogicalTX batch artifact. Any number of properties may be specified.

Syntax:

<properties>
    <property name="{*property-name*}" value="{ *name-value*}" />
</properties>

Where:

name	Specifies a unique property name. It must be a valid XML string value. It should match a property name specified in a corresponding batch artifact. This is a required attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See section [REF] for further information on substitution symbols.

4.2.6.4 SubJob Collector

A SubJob Collector is useful for sending intermediary results for analysis from each partition to the step's SubJob Analyzer. A separate SubJob Collector instance runs on each thread executing a partition of the step. The collector is invoked at the conclusion of each checkpoint for chunking type steps; it is invoked once at the end for batchlet type steps. A collector returns a Java Externalizable object, which is delivered to the split's SubJob Analyzer. See section [REF] for further information about the SubJob Analyzer. The collector element specifies a reference to a SubJobCollector batch artifact; see section [REF] for further information.

Syntax:

<collector ref="{*name*}">

Where:

ref	Specifies the name of a batch artifact, as known to the batch artifact loader. See section [REF] for further information about batch artifact names and batch artifact loading.

4.2.6.4.1 SubJob Collector Properties

The 'properties' element may be specified as a sub-element of the collector element. It is used to pass property values to a SubJobCollector batch artifact. Any number of properties may be specified.

Syntax:

<properties>
    <property name="{*property-name*}" value="{ *name-value*}" />
</properties>

Where:

name	Specifies a unique property name. It must be a valid XML string value. It should match a property name specified in a corresponding batch artifact. This is a required attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See section [REF] for further information on substitution symbols.

4.2.6.5 SubJob Analyzer

A SubJob Analyzer receives intermediary results from each partition sent via the step's SubJob Collector. A subjob analyzer runs on the step main thread and serves as a collection point for this data. The SubJobAnalyzer also receives control with the partition exit status for each partition, after that partition ends. An analyzer can be used to implement custom exit status handling for the step, based on the results of the individual partitions. The analyzer element specifies a reference to a SubJobAnalyzer batch artifact; see section [REF] for further information.

Syntax:

<analyzer ref="{*name*}">

Where:

ref	Specifies the name of a batch artifact, as known to the batch artifact loader. See

	section [REF] for further information about batch artifact names and batch artifact loading.

4.2.6.5.1 SubJob Analyzer Properties

The 'properties' element may be specified as a sub-element of the analyzer element. It is used to pass property values to a SubJobAnalyzer batch artifact. Any number of properties may be specified.

Syntax:

<properties>
<property name="{*property-name*}" value="{ *name-value*}" />
</properties>

Where:

name	Specifies a unique property name. It must be a valid XML string value. It should match a property name specified in a corresponding batch artifact. This is a required attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See section [REF] for further information on substitution symbols.

4.3 Flow

A flow defines a sequence of steps that execute together as a unit. When the flow is finished, it is the entire flow that transitions to the next execution element. A flow may transition to a step, split, decision, or another flow. The steps within a flow may only transition among themselves; steps in a flow may not transition to elements outside of the flow. Besides steps, a flow may contain decision elements (see section [REF]). A flow may not contain a split; however, a split may contain a flow. See section [REF] for more on splits.

Syntax:

<flow id="*{name}*"next="*{ flow-id|step-id|split-id|decision-id}*" abstract="*{<u>true</u>|false}*" parent="*{name}*">
<step> ... </step> ...
</flow>

Where:

id	Specifies the logical *name* of the flow and is used for identification purposes. It must be a valid XML string value. This is a required attribute.

next	Specifies the next step, flow, split, or decision to run after this step is complete. It must be a valid XML string value. This is an optional attribute. The default is false
abstract	Specifies whether or not this step is an abstract step. Abstract step are used for step inheritance through the parent attribute. The elements of an abstract step are combined with the elements of the inheriting step.
parent	Specifies the id value of a step definition to be used by the current step.

4.3.1 Flow Partitioning

Similar to a step, a flow may also be partitioned. When a flow is partitioned, all steps defined to that flow execute in sequence for each partition. For example, if a flow defined step1, followed by step2 and was configured into a 2-way partition, then 2 separate threads would each execute the sequence step1, followed by step2.

Syntax:

<partition instances="{*number*}"/>

Where:

instances	Specifies the number of partition instances (threads) for this partitioned flow. This is a an optional attribute. The default is 1.

Example:

The following Job XML snippet shows how to specify a flow partitioned onto 2 threads:

```
<flow id="Flow1">
        <step id="Step1" parent="Concrete.Step1" />
        <step id="Step2" parent="Concrete.Step2" />
        <partition instances="2"/>
</step>
```

4.3.1.1 Partition Properties

When defining a statically partitioned flow, it is possible to specify unique property values to pass to each partition directly in the Job XML using the property element.

Syntax:

```
<properties>
        <property name="{property-name}.${partition-number}" value="{ name-value}" />
```

```
</properties>
```

Where:

name	Specifies a unique property name and partition number. It must be a valid XML string value. It should match a property name specified in a corresponding batch artifact. This is a required attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See section [REF] for further information on substitution symbols.

Example:

The following Job XML snippet shows a 2-way partition (i.e. 2 threads) with a unique value for the property named "filename" for each partition:

```
<partition instances="2">
    <properties>
        <property name="filename.1" value="/tmp/file1.txt"/>
        <property name="filename.2" value="/tmp/file2.txt"/>
    </properties>
</partition
```

The "target" attribute may be specified on the property element to apply that property to a specific batch element on a specific step if there is more than one possible target batch artifact.

E.g.

```
<flow id="flow1">
    <step id="Step1" parent="Parent.Step1" >
        <properties merge="false">
            <property name="filename" value="${step1.reader.filename}" target="reader">
            <property name="filename" value="${step1.writer.filename}" target="writer">
        </properties>
    </step>
    <step id="Step2" parent="Parent.Step2" >
        <properties merge="false">
            <property name="filename" value="${step2.filename}">
        </properties>
    </step>
    <partition instances="2">
        <properties>
            <property name="step1.reader.filename.1" value="/tmp/infile1.txt">
```

```
                    <property name="step1.writer.filename.1" value="/tmp/outfile1.txt">
                    <property name="step1.reader.filename.2" value="/tmp/infile2.txt">
                    <property name="step1.writer.filename.2" value="/tmp/outfile2.txt">
                    <property name="step2.filename.1" value="/tmp/outfile3.txt">
                    <property name="step2.filename.2" value="/tmp/outfile3.txt" >
            </properties>
      </partition>
</flow>
```

4.3.1.2 Partition Algorithm

The partition algorithm provides a programmatic means for calculating the number of partitions for a partitioned flow. The partition algorithm also specifies the properties for each partition. The use of a partition algorithm overrides the 'instances' attribute on the parent partition element, and also overrides any batch properties specified on that same element. The algorithm element specifies a reference to a PartitionAlgorithm batch artifact; see section [REF] for further information.

Syntax:

<algorithm ref="{*name*}">

Where:

ref	Specifies the name of a batch artifact, as known to the batch artifact loader. See section [REF] for further information about batch artifact names and batch artifact loading.

4.3.1.2.1 Algorithm Properties

The 'properties' element may be specified as a sub-element of the algorithm element. It is used to pass property values to a PartitionAlgorithm batch artifact. Any number of properties may be specified.

Syntax:

<properties>
<property name="{*property-name*}" value="{ *name-value*}" />
</properties>

Where:

name	Specifies a unique property name. It must be a valid XML string value. It should match

	a property name specified in a corresponding batch artifact. This is a required attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See section [REF] for further information on substitution symbols.

4.3.1.3 Logical Transaction

A partitioned flow executes in the context of a logical transaction. A logical transaction provides a kind of unit of work demarcation around the execution of the collection of threads. Programmatic interception of the logical transaction lifecycle is possible by providing a logical transaction. The logicalTX element specifies a reference to a LogicalTX batch artifact; see section [REF] for further information.

Syntax:

<logicalTX ref="{*name*}">

Where:

ref	Specifies the name of a batch artifact, as known to the batch artifact loader. See section [REF] for further information about batch artifact names and batch artifact loading.

4.3.1.3.1 LogicalTX Properties

The 'properties' element may be specified as a sub-element of the logicalTX element. It is used to pass property values to a LogicalTX batch artifact. Any number of properties may be specified.

Syntax:

```
<properties>
        <property name="{property-name}" value="{ name-value}" />
</properties>
```

Where:

name	Specifies a unique property name. It must be a valid XML string value. It should match a property name specified in a corresponding batch artifact. This is a required attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See

	section [REF] for further information on substitution symbols.

4.3.1.4 SubJob Collector

A SubJob Collector is useful for sending intermediary results for analysis from each partition to the flow's SubJob Analyzer. The flow collector receives control for all steps belonging to the flow. A separate SubJob Collector instance runs on each thread executing a partition of the flow. For a non-partitioned step, the collector is invoked at the conclusion of each checkpoint for chunking type steps; it is invoked once at the end for batchlet type steps. If the step is partitioned, the split collector is invoked once only at the end of the entire step. A collector returns a Java Externalizable object, which is delivered to the split's SubJob Analyzer. See section [REF] for further information about the SubJob Analyzer. The collector element specifies a reference to a SubJobCollector batch artifact; see section [REF] for further information.

Syntax:

<collector ref="{*name*}">

Where:

ref	Specifies the name of a batch artifact, as known to the batch artifact loader. See section [REF] for further information about batch artifact names and batch artifact loading.

4.3.1.4.1 SubJob Collector Properties

The 'properties' element may be specified as a sub-element of the collector element. It is used to pass property values to a SubJobCollector batch artifact. Any number of properties may be specified.

Syntax:

<properties>
    <property name="{*property-name*}" value="{ *name-value*}" />
</properties>

Where:

name	Specifies a unique property name. It must be a valid XML string value. It should match a property name specified in a corresponding batch artifact. This is a required

	attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See section [REF] for further information on substitution symbols.

4.3.1.5 SubJob Analyzer

A SubJob Analyzer receives intermediary results from each partition sent via the flow's SubJob Collector. A subjob analyzer runs on the flow main thread and serves as a collection point for this data. The SubJobAnalyzer also receives control with the partition exit status for each partition, after that partition ends. An analyzer can be used to implement custom exit status handling for the step, based on the results of the individual partitions. The analyzer element specifies a reference to a SubJobAnalyzer batch artifact; see section [REF] for further information.

Syntax:

<analyzer ref="{*name*}">

Where:

ref	Specifies the name of a batch artifact, as known to the batch artifact loader. See section [REF] for further information about batch artifact names and batch artifact loading.

4.3.1.5.1 SubJob Analyzer Properties

The 'properties' element may be specified as a sub-element of the analyzer element. It is used to pass property values to a SubJobAnalyzer batch artifact. Any number of properties may be specified.

Syntax:

<properties>
    <property name="{*property-name*}" value="{ *name-value*}" />
</properties>

Where:

name	Specifies a unique property name. It must be a valid XML string value. It should match a property name specified in a corresponding batch artifact. This is a required attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See

	section [REF] for further information on substitution symbols.

4.4 Split

A split defines a set of flows that execute concurrently. See section [REF] for more on flows. Each flow runs on a separate thread. The split is finished after all flows complete. When the split is finished, it is the entire split that transitions to the next execution element. A split may transition to a step, split, decision, or another flow. The flows within a split may only transition among themselves; flows in a split may not transition to elements outside of the split. Besides flows, a split may contain decision elements (see section [REF]). A split may not contain a split.

Syntax:

<split id="{*name*}"next="{*flow-id|step-id|split-id|decision-id*}" abstract="{*true|false*}" parent="{*name*}">
<flow> ... </flow> ...
</flow>

Where:

id	Specifies the logical *name* of the split and is used for identification purposes. It must be a valid XML string value. This is a required attribute.
next	Specifies the next step, flow, split, or decision to run after this step is complete. It must be a valid XML string value. This is an optional attribute. The default is false
abstract	Specifies whether or not this step is an abstract step. Abstract step are used for step inheritance through the parent attribute. The elements of an abstract step are combined with the elements of the inheriting step.
parent	Specifies the id value of a step definition to be used by the current step.

4.4.1 Split Concurrency

A split, by its very nature, runs multiple flows on concurrent threads, one flow per thread. Similar to a partitioned step or flow, a split may require additional control over the execution of its parallel parts. As with partitioning, each thread is treated as a subjob. A logical transaction demarcates the execution of these subjobs. A logical transaction batch artifact can be attached to the split to allow control over this logical transaction, allowing for execution of compensation logic if necessary in the event of failure of one or more flows. Similarly, subjob collector/analyzer batch artifacts may be added to the split to allow greater control over formulating exit status for the split. The concurrency element may be nested inside a split to provide collection of these concurrency control artifacts for a splitSyntax:

<concurrency/>

4.4.1.1 Logical Transaction

A split executes in the context of a logical transaction. A logical transaction provides unit of work demarcation around the execution of the collection of threads. Programmatic interception of the logical transaction lifecycle is possible by providing a logical transaction. The logicalTX element specifies a reference to a LogicalTX batch artifact; see section [REF] for further information.

Syntax:

<logicalTX ref="{*name*}">

Where:

ref	Specifies the name of a batch artifact, as known to the batch artifact loader. See section [REF] for further information about batch artifact names and batch artifact loading.

Example:

```
<split id="Split1">
        <flow id="Flow1" parent="Concrete.Flow1" next="Flow2"/>
        <flow id="Flow2" parent="Concrete.Flow2" />
        <concurrency>
                <logicalTX ref="MySplitLogicalTX"/>
        </concurrency>
</split>
```

4.4.1.1.1 LogicalTX Properties

The 'properties' element may be specified as a sub-element of the logicalTX element. It is used to pass property values to a LogicalTX batch artifact. Any number of properties may be specified.

Syntax:

```
<properties>
        <property name="{property-name}" value="{ name-value}" />
</properties>
```

Where:

name	Specifies a unique property name. It must be a valid XML string value. It should match

	a property name specified in a corresponding batch artifact. This is a required attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See section [REF] for further information on substitution symbols.

4.4.1.2 SubJob Collector

A SubJob Collector is useful for sending intermediary results for analysis from each flow to the split's SubJob Analyzer. The split collector receives control for all steps belonging to each flow in the split. A separate SubJob Collector instance runs on each thread executing a flow of a split. For a non-partitioned flow, the collector is invoked at the conclusion of each checkpoint for chunking type steps; it is invoked once at the end for batchlet type steps. If the flow is partitioned, the split collector is invoked once only at the end of the entire flow. A collector returns a Java Externalizable object, which is delivered to the split's SubJob Analyzer. See section [REF] for further information about the SubJob Analyzer. The collector element specifies a reference to a SubJobCollector batch artifact; see section [REF] for further information.

Syntax:

<collector ref="{*name*}">

Where:

ref	Specifies the name of a batch artifact, as known to the batch artifact loader. See section [REF] for further information about batch artifact names and batch artifact loading.

Example:

```
<split id="Split1">
	<flow id="Flow1" parent="Concrete.Flow1" next="Flow2"/>
	<flow id="Flow2" parent="Concrete.Flow2" />
	<concurrency>
		<collector ref="MySplitCollector"/>
	</concurrency>
</split>
```

4.4.1.2.1 SubJob Collector Properties

The 'properties' element may be specified as a sub-element of the collector element. It is used to pass property values to a SubJobCollector batch artifact. Any number of properties may be specified.

Syntax:

```
<properties>
	<property name="{property-name}" value="{ name-value}" />
</properties>
```

Where:

name	Specifies a unique property name. It must be a valid XML string value. It should match a property name specified in a corresponding batch artifact. This is a required attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See section [REF] for further information on substitution symbols.

4.4.1.3 SubJob Analyzer

A SubJob Analyzer receives intermediary results from each flow sent via the split's SubJob Collector. A subjob analyzer runs on the split main thread and serves as a collection point for this data. The SubJobAnalyzer also receives control with the flow exit status for each flow, after the flow ends. An analyzer can be used to implement custom exit status handling for the split, based on the results of the individual flows. The analyzer element specifies a reference to a SubJobAnalyzer batch artifact; see section [REF] for further information.

Syntax:

```
<analyzer ref="{name}">
```

Where:

ref	Specifies the name of a batch artifact, as known to the batch artifact loader. See section [REF] for further information about batch artifact names and batch artifact loading.

Example:

```
<split id="Split1">
	<flow id="Flow1" parent="Concrete.Flow1" next="Flow2"/>
	<flow id="Flow2" parent="Concrete.Flow2" />
	<concurrency>
		<analyzer ref="MySplitAnalyzer"/>
	</concurrency>
```

</split>

4.4.1.3.1 SubJob Analyzer Properties

The 'properties' element may be specified as a sub-element of the analyzer element. It is used to pass property values to a SubJobAnalyzer batch artifact. Any number of properties may be specified.

Syntax:

```
<properties>
	<property name="{property-name}" value="{ name-value}" />
</properties>
```

Where:

name	Specifies a unique property name. It must be a valid XML string value. It should match a property name specified in a corresponding batch artifact. This is a required attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See section [REF] for further information on substitution symbols.

4.5 Batch and Exit Status

Batch execution reflects a sequence of state changes, culminating in an end state after a job has terminated. These state changes apply to entire job as a whole, as well as to each step within the job. These state changes are exposed through the programming model as status values. There is both a runtime status value, called "batch status", as well as a user-defined value, called "exit status".

Each step, flow, split, and job ends with a batch status and exit status value. Batch status is set by the batch runtime; exit status may be set through the Job XML or by the batch application. By default, the exit status is the same as the batch status. The batch and exit status values are available in the JobContext and StepContext runtime objects. The overall batch and exit status for the job are available through the JobOperator interface. Batch and exit status values are strings. The following batch status values are defined:

Value	Meaning
STARTING	Batch job has been passed to the batch runtime for execution through the JobOperator interface start or restart operation. A step has a status of STARTING before it actually begins execution.

STARTED	Batch job has begun execution by the batch runtime. A step has a status of STARTED once it has begun execution.
STOPPING	Batch job has been requested to stop through the JobOperator interface stop operation or by a <stop> element in the Job XML. A step has a status of STOPPING as soon as stop processing has commenced by the batch runtime.
STOPPED	Batch job has been stopped through the JobOperator interface stop operation or by a <stop> element in the Job XML. A step has a status of STOPPED once it has actually been stopped by the batch runtime.
FAILED	Batch job has ended due to an unresolved exception or by a <fail> element in the Job XML. A step has a status of FAILED under the same conditions.
COMPLETED	Batch job has ended normally or by an <end> element in the Job XML. A step has a status of COMPLETED under the same conditions.

A job will finish execution under the following conditions:

1. A job-level execution element (step, flow, or split) that does not specify a "next" attribute finishes execution. In this case, the batch status is set to COMPLETED.

2. A step throws an unresolved exception. In this case, the batch status is set to FAILED. In the case of partitioned or concurrent (split) step execution, all other still-running parallel instances end with STOPPED batch status.

3. A step or decision (see section [REF]) terminates execution with a stop, end, or fail element. In this case, the batch status is STOPPED, COMPLETED, or FAILED, respectively. In the case of partitioned or concurrent (split) step execution, all other still-running parallel instances end with STOPPED batch status. See following sections [REF] and [REF] for more on these element types.

4.5.1 Batch and Exit Status for Steps

Step batch status is set by the batch runtime at the conclusion of the step. Step exit status may be set by any batch artifact configured to the step by invoking the exit status setter method in the StepContext object. See section [REF] for further information about the StepContext object.

4.5.1.1 Step termination may also be configured in the Job XML using the fail, end, and stop elements. See the following sections for more about those element types.Fail Element

The fail element is used to terminate the job at the current step. The batch status is set to FAILED. The job exit status may be optionally set; if set, this overrides the job exit status value in the JobContext.

Syntax:

<fail on="{*exit status*}" exit-status="{*exit status*}"/>

Where:

on	Specifies the exit status value that activates this fail element. It must be a valid XML string value. Wildcards of "*" and "?" may be used. "*" matches one or more characters. "?" matches exactly one character. It must match an exit status value in order to have effect. This is a required attribute.
exit-status	Specifies the exit status for the job. It must be a valid XML string value. This is an optional attribute. If not specified, the job exit status is the exit status value from the JobContext, which defaults to the job batch status value.

Example:

```
<step id="Step1">
        <fail on="FAILED" exit-status="EARLY COMPLETION">
</step>
```

4.5.1.2 End Element

The end element is used to terminate the job at the current step. The batch status is set to COMPLETED. The job exit status may be optionally set; if set, this overrides the job exit status value in the JobContext.

Syntax:

<end on="{*exit status*}" exit-status="{*exit status*}"/>

Where:

on	Specifies the exit status value that activates this end element. It must be a valid XML string value. Wildcards of "*" and "?" may be used. "*" matches one or more characters. "?" matches exactly one character. It must match an exit status value in order to have effect. This is a required attribute.
exit-status	Specifies the exit status for the job. It must be a valid XML string value. This is an optional attribute. If not specified, the job exit status is the exit status value from the JobContext, which defaults to the job batch status value.

Example:

```
<step id="Step1">
        <end on="COMPLETED" exit-status="EARLY COMPLETION">
</step>
```

4.5.1.3 Stop Element

The stop element is used to terminate the job at the current step. The batch status is set to STOPPED. The job exit status may be optionally set; if set, this overrides the job exit status value in the JobContext. The stop element may also optionally set the job-level step, flow, or split at which the job will restart once restarted.

<stop on="{*exit status*}" exit-status="{*exit status*}" restart="{*step id* | *flow id* | *split id*}"/>

Where:

on	Specifies the exit status value that activates this end element. It must be a valid XML string value. Wildcards of "*" and "?" may be used. "*" matches one or more characters. "?" matches exactly one character. It must match an exit status value in order to have effect. This is a required attribute.
exit-status	Specifies the exit status for the job. It must be a valid XML string value. This is an optional attribute. If not specified, the job exit status is the exit status value from the JobContext, which defaults to the job batch status value.
restart	Specifies the job-level step, flow, or split on which to restart when the job is restarted. It must be a valid XML string value. This is an optional attribute. By default it is the current step.

Example:

```
<step id="Step1">
        <stop on="COMPLETED" restart="step2"/>
</step>
```

4.5.2 Batch and Exit Status for Flows

A flow finishes when its final step finishes. The final step in a flow is a step with no "next" attribute. A flow may also finish when any of its steps finishes due to an unhandled exception, or due to an end, fail, or fail element. In the case of partitioned flow, all other still-running parallel instances end with STOPPED batch status. A flow also has a batch and exit status. The batch and exit status of a flow is the

batch and exit status of the last step to execute. The batch and exit status of a flow is available in the FlowContext. See section [REF] for FlowContext.

4.5.3 Batch and Exit Status for Splits

A split finishes when its final flow finishes. The final flow in a split is a flow with no "next" attribute. A split may also finish when any of its flows finishes due to an unhandled exception, or due to an end, fail, or fail element by any of its constituent steps. All other still-running parallel flows end with STOPPED batch status. A split also has a batch and exit status. The batch and exit status of a split is the batch and exit status of the last flow to execute. The batch and exit status of a split is available in the SplitContext. See section [REF] for SplitContext.

4.6 Decision

A decision provides a customized way of determining sequencing among steps, flows, and splits. The decision element is a job-level element. A job may contain any number of decision elements. A decision element is the target of the "next" attribute from a split or from a job-level step or flow. A decision must supply a decider batch artifact (see section [REF]). The decider's purpose is to decide the next transition. The decision uses stop, fail, end, and next elements to select the next transition. The decider may set a new exit status to facilitate the transition choice.

The decider has access to JobContext and to the context of last job-level execution element that finished. That means a StepContext is available if the decision was transitioned to from a step; a FlowContext if from a flow; and a SplitContext if from a split. The context objects hold the batch and exit status values for the last execution element that finished. The decider may use this context information to make its decision.

Syntax:

<decision id="{*name}*" decider="{*ref-name*}">

Where:

id	Specifies the logical *name* of the decision and is used for identification purposes. It must be a valid XML string value. This is a required attribute.
ref	Specifies the *name* of a batch artifact, as known to the batch artifact loader. See section [REF] for further information about batch artifact names and batch artifact loading.

Example:

```
<decision id="AfterFlow1" decider="MyDecider" >
...
</decision>
```

4.6.1 Fail Element

The fail element is a sub-element of the decision element. A decision element may contain any number of fail elements. The fail element is used to terminate the job at the current decision. The batch status is set to FAILED. The job exit status may be optionally set; if set, this overrides the job exit status value in the JobContext.

Syntax:

<fail on="{*exit status*}" exit-status="{*exit status*}"/>

Where:

On	Specifies the exit status value that activates this fail element. It must be a valid XML string value. Wildcards of "*" and "?" may be used. "*" matches one or more characters. "?" matches exactly one character. It must match an exit status value in order to have effect. This is a required attribute.
exit-status	Specifies the exit status for the job. It must be a valid XML string value. This is an optional attribute. If not specified, the job exit status is the exit status value from the JobContext, which defaults to the job batch status value.

Example:

```
<decision id="AfterFlow1" decider="MyDecider" >
        <fail on="FAILED" exit-status="DO NOT RESTART"/>
</decision>
```

4.6.2 End Element

The end element is a sub-element of the decision element. A decision element may contain any number of end elements. The end element is used to terminate the job at the current decision. The batch status is set to COMPLETED. The job exit status may be optionally set; if set, this overrides the job exit status value in the JobContext.

Syntax:

<end on="{*exit status*}" exit-status="{*exit status*}"/>

Where:

on	Specifies the exit status value that activates this end element. It must be a valid XML string value. Wildcards of "*" and "?" may be used. "*" matches one or more characters. "?" matches exactly one character. It must match an exit status value in order to have effect. This is a required attribute.
exit-status	Specifies the exit status for the job. It must be a valid XML string value. This is an optional attribute. If not specified, the job exit status is the exit status value from the JobContext, which defaults to the job batch status value.

Example:

```
<decision id="AfterFlow1" decider="MyDecider" >
	<end on="COMPLETED" exit-status="EARLY COMPLETION"/>
</decision>
```

4.6.3 Stop Element

The stop element is a sub-element of the decision element. A decision element may contain any number of stop elements. The stop element is used to terminate the job at the current decision. The batch status is set to STOPPED. The job exit status may be optionally set; if set, this overrides the job exit status value in the JobContext. The stop element may also optionally set the job-level step, flow, or split at which the job will restart once restarted.

<stop on="{*exit status*}" exit-status="{*exit status*}" restart="{*step id* | *flow id* | *split id*}"/>

Where:

on	Specifies the exit status value that activates this end element. It must be a valid XML string value. Wildcards of "*" and "?" may be used. "*" matches one or more characters. "?" matches exactly one character. It must match an exit status value in order to have effect. This is a required attribute.
exit-status	Specifies the exit status for the job. It must be a valid XML string value. This is an optional attribute. If not specified, the job exit status is the exit status value from the JobContext, which defaults to the job batch status value.
restart	Specifies the job-level step, flow, or split on which to restart when the job is restarted. It must be a valid XML string value. This is an optional attribute. By default it is the current step.

Example:

```
<decision id="AfterFlow1" decider="MyDecider" >
    <stop on="COMPLETED" exit-status="RESTART" restart="Step2"/>
</decision>
```

4.6.4 Next Element

The next element is a sub-element of the decision element. A decision element may contain any number of next elements. The next element is used to transition the current decision to the next execution element.

<next on="{*exit status*}" to="{*step id* | *flow id* | *split id*}"/>

Where:

on	Specifies the exit status value that activates this end element. It must be a valid XML string value. Wildcards of "*" and "?" may be used. "*" matches one or more characters. "?" matches exactly one character. It must match an exit status value in order to have effect. This is a required attribute.
to	Specifies the job-level step, flow, or split to which to transition after this decision. It must be a valid XML string value. This is a required attribute.

Example:

```
<decision id="AfterFlow1" decider="MyDecider" >
    <next on="*" to="Step2" />
</decision>
```

4.6.5 Decision Properties

The 'properties' element may be specified as a sub-element of the decision element. It is used to pass property values to a decider. Any number of properties may be specified.

Syntax:

<properties> <property name="{*property-name*}" value="{ *name-value*}" />
</properties>Where:

name	Specifies a unique property name. It must be a valid XML string value. It should match a property name specified in a corresponding batch artifact. This is a required attribute.
value	Specifies the value corresponding to the named name. It must be a valid XML string value. This is a required attribute. The value may be a substitution symbol. See

	section [REF] for further information on substitution symbols.

4.7 Property Substitution and Overrides

The Job XML model supports property specification at multiple levels:

1. job level
2. step level
3. batch artifact level (e.g. ItemReader, ItemWriter, Batchlet, listeners, etc)

Properties may also be specified as Job Parameters (see section [REF]). Job XML implements an override scheme and supports property substitution.

4.7.1 Property Overrides

Since properties can be specified at multiple levels, there is a hierarchy. When the same named property is specified within the hierarchy, a precedence rule determines which property instance establishes the effective property value.

The precedence rule is:

1. A property specified as a Job Parameter overrides all occurrences of the same-named property in Job XML and in System Properties. Note: however, a Job Parameter property does not change the actual System Properties pool.
2. A property defined as a System property overrides all occurrences of the same-named property in Job XML.
3. Within Job XML, a property specified for a particular XML element overrides all occurrences of the same named property in nested elements. E.g. job properties override step properties; step properties override chunk properties; etc

4.7.1.1 Override from Job Parameters Example

Note: code is stylized for illustrative purposes only.

```
Properties props= new Properties();
props.setProperty("outfile","/usr/local/tmp/job1.outfile.txt");
JobOperator.start("Job1",props);
```

Effective value of outfile for Step1 batchlet is "/usr/local/tmp/job1.outfile.txt" because job parameters override batchlet level property.

```
<job id="Job1">
	<step id="Step1">
		<batchlet ref="MyBatchlet">
			<properties>
				<property name="outfile" value="/tmp/outfile.txt" />
			</properties>
		</batchlet>
	</step>
</job>
```

4.7.1.2 Override from JVM Properties Example

JVM properties override Job XML properties, not other way around, so value of ${file.separator} is "/" or "/" depending on platform.

```
<job id="Job1">
	<properties>
		<property name="file.separator" value="@" />
	</properties>
	<step id="Step1">
		<batchlet ref="MyBatchlet">
			<properties>
				<property name="outfile"
					value="${file.separator}tmp${file.separator}outfile.txt" />
			</properties>
		</batchlet>
	</step>
</job>
```

4.7.1.3 Override from Job XML Example

Effective value of outfile for Step1 batchlet is "/usr/local/tmp/job1.outfile.txt" because job level property overrides batchlet level property.

```
<job id="Job1">
	<properties>
		<property name="outfile" value="/usr/local/tmp/job1.outfile.txt" />
```

```
        </properties>
        <step id="Step1">
                <batchlet ref="MyBatchlet">
                        <properties>
                                <property name="outfile" value="/tmp/outfile.txt" />
                        </properties>
                </batchlet>
        </step>
</job>
```

4.7.1.4 Override with Inheritance Example

Effective value of outfile for Step1 batchlet is "/usr/local/tmp/job1.outfile.txt" because job level property overrides batchlet level property. This is really no different than "Override from Job XML" base.

```
<step id="Parent.Step1">
        <batchlet ref="MyBatchlet">
                <properties>
                        <property name="outfile" value="/tmp/outfile.txt" />
                </properties>
        </batchlet>
</step>

<job id="Job1">
        <properties>
                <property name="outfile" value="/usr/local/tmp/job1.outfile.txt" />
        </properties>
        <step id="Step1" parent="Parent.Step1"/>
</job>
```

4.7.1.5 Override with Partitions Example

Effective value of outfile for Step1 batchlet is "/tmp/outfile1.txt" for partition 1 and "/tmp/outfile2.txt" for partition 2.

```
<job id="Job1">
        <step id="Step1">
                <partition instances="2"/>
```

```
                <properties>
                        <!--
                        Partition properties are job parameters for subjobs.  Syntax is
                        property.N where N is the partition number.  The "N" gets stripped
                         off as the property is passed to the partition.
                        -->
                        <property name="outfile.1" value="/tmp/outfile1.txt" />
                        <property name="outfile.2" value="/tmp/outfile2.txt" />
                </properties>
            </partition>
            <batchlet ref="MyBatchlet">
                <properties>
                        <property name="outfile" value="/usr/local/tmp/job1.outfile.txt" />
                </properties>
            </batchlet>
        </step>
</job>
```

4.7.2 Property Substitution

Job XML supports property substitution using ${} operator. The substitution operator may be used in attribute values anywhere in Job XML except on the parent attribute, which is explained further down in this paragraph. The ${} operator must specify a property by name. All substitutions are resolved before the job executes. Job XML inheritance is resolved first, then substitutions. For this reason substitution cannot be used on the parent attribute. Substitution always resolves to the property value determined by the override precedence rules, as defined in the preceding section.

4.7.2.1 Substitution from Job Parameters Example

Note: code is stylized for illustrative purposes only.

```
Properties props= new Properties();
props.setProperty("step1.outfile","/usr/local/tmp/job1.outfile.txt");
JobOperator.start("Job1",props);
```

Effective value of outfile for Step1 batchlet is "/usr/local/tmp/job1.outfile.txt" because job parameters provided value of property "step1.outflie".

```
<job id="Job1">
	<step id="Step1">
		<batchlet ref="MyBatchlet">
			<properties>
				<property name="outfile" value="${step1.outfile}" />
			</properties>
		</batchlet>
	</step>
</job>
```

4.7.2.2 Substitution from System Properties Example

Resolved value of property "outfile" is "/tmp/outfile.txt".

```
<job id="Job1">
	<step id="Step1">
		<batchlet ref="MyBatchlet">
			<properties>
				<property name="outfile"
					value="${file.separator}tmp${file.separator}outfile.txt" />
			</properties>
		</batchlet>
	</step>
</job>
```

4.7.2.3 Substitution from Job XML Example

Resolved value of property "outfile" is "/tmp/outfile.txt".

```
<job id="Job1">
	<properties>
		<property name="step1.outfile" value="/tmp/outfile.txt" />
	</properties>
	<step id="Step1">
		<batchlet ref="MyBatchlet">
			<properties>
				<property name="outfile" value="${step1.outfile}" />
			</properties>
		</batchlet>
```

```
	</step>
</job>
```

4.7.2.4 Substitution with Inheritance Example

Resolved value of property "outfile" is "/tmp/outfile.txt". This is really no different than "Substitution from Job XML" base.

```
<step id="Parent.Step1">
	<batchlet ref="MyBatchlet">
		<properties>
			<property name="outfile" value="${parent.outfile}" />
		</properties>
	</batchlet>
</step>

<job id="Job1" parent="Parent.Step1">
	<properties>
		<property name="parent.outfile" value="/tmp/outfile.txt" />
	</properties>
	<step id="Step1" parent="Parent.Step1"/>
</job>
```

4.7.2.5 Substitution with Partitions Example

Resolved value of property "outfile" is "/tmp/outfile1.txt" for partition 1 and "/tmp/outfile2.txt" for partition 2.

```
<job id="Job1">
	<step id="Step1">
		<partition instances="2"/>
			<properties>
				<!--
				Partition properties are job parameters for subjobs.  Syntax is
				property.N where N is the partition number.  The "N" gets stripped
				 off as the property is passed to the partition.
				-->
				<property name="step1.outfile.1" value="/tmp/outfile1.txt" />
				<property name="step1.outfile.2" value="/tmp/outfile2.txt" />
```

```
                </properties>
            </partition>
            <batchlet ref="MyBatchlet">
                <properties>
                    <property name="outfile" value="${step1.outfile}" />
                </properties>
            </batchlet>
    </step>
</job>
```

4.8 Job XML Inheritance

Job XML supports inheritance of job, step, flow, and split elements. A child element can inherit only from a like parent element - e.g. job can inherit only job; step can inherit only step, etc.. Inheritance works by combining the sub-elements of the parent with that of the child; the parent element is completely overridden by the child - e.g. the child job element completely overrides the parent.

Example:

parent-job.xml:

```
<job id="step.auditor">
    <listeners>
        <listener ref="StepAuditor"/>
    </listeners>
</step>
```

child-job.xml:

```
<job id="Job1" parent="step.auditor"/>
    <step id="step1">
        <chunk reader="MyReader" processor="MyProcessor" writer="MyWriter/>
    </step>
</job>
```

When "Job1" is started, the effective Job XML:

```
<job id="Job1"/>
    <listeners>
        <listener ref="StepAuditor"/>
    </listeners>
```

```
	<step id="step1">
		<chunk reader="MyReader" processor="MyProcessor" writer="MyWriter/>
	</step>
</job>
```

Not only are the child's sub-elements combined with that of the parent's, but the attributes contained by those sub-elements are also combined, as illustrated in the following example:

parent-step.xml:

```
<step id="step.auditor" abstract="true">
	<step id="MyProcessor">
		<chunk processor="MyProcessor">
			<properties>
				<property name="audit" value="true"/>
			</properties>
		</chunk>
	</step>
</step>
```

Note: abstract="true" must be specified if the parent element does not include all required elements and attributes.

child-job.xml:

```
<job id="Job2" />
	<step id="step1" parent="MyProcessor">
		<chunk reader="MyReader" writer="MyWriter/>
	</step>
</job>
```

When "Job2" is started, the effective Job XML:

```
<job id="Job2"/>
	<step id="step1">
		<chunk reader="MyReader" processor="MyProcessor" writer="MyWriter>
			<properties>
				<property name="audit" value="true"/>
			</properties>
		</chunk>
```

```
        </step>
</job>
```

4.8.1 Merging Lists

When list elements are included in the inheritance process, there is a choice of overriding the parent list or merging with it.   All list elements support this capability.

Example:

parent-step.xml:

```
<step id="step.auditor" abstract="true">
        <step id="MyProcessor">
                <chunk processor="MyProcessor">
                        <properties>
                                <property name="audit" value="true"/>
                        </properties>
                </chunk>
        </step>
</step>
```

child-job.xml:

```
<job id="Job3" />
        <step id="step1" parent="MyProcessor">
                <chunk reader="MyReader" writer="MyWriter>
                        <properties merge="true">
                                <property name="infile" value="/tmp/input.txt" target="reader"/>
                                <property name="outfile" value="/tmp/output.txt" target="writer"/>
                        </properties>
                </chunk>
        </step>
</job>
```

When "Job3" is started, the effective Job XML:

```
<job id="Job3"/>
        <step id="step1">
                <chunk reader="MyReader" processor="MyProcessor" writer="MyWriter>
                        <properties>
```

```
                <property name="audit" value="true"/>
                <property name="infile" value="/tmp/input.txt" target="reader"/>
                <property name="outfile" value="/tmp/output.txt" target="writer"/>
            </properties>
        </chunk>
    </step>
</job>
```

4.8.2 Inheritance Namespace

Top-level job, step, flow, and split id= values must be unique across all files found in the batch runtime's specified "jobpath". See section [ref] for more information on the jobpath configuration.

5 Batch Programming Model

5.1 Steps

A batch step is either chunk or batchlet.

5.1.1 Chunk

A chunk type step performs item-oriented processing using a reader-processor-writer batch pattern and does checkpointing.

5.1.1.1 @ItemReader

The @ItemReader annotation identifies a class as an item reader, which is one of the three artifact types that comprise a chunk-type step.

Syntax:

@ItemReader[("<id>")]

Where:

<id>	is the optional name for the item reader. The default is the item reader class name.

Example:

@ItemReader("MyItemReader")

```
public class MyItemReaderImpl {...}
```

5.1.1.1.1 @Open

The @Open annotation identifies a method that does initialization processing for an item reader. This method receives a Java Externalizable that contains the last checkpoint position for the reader. If the checkpoint is null, this method should position the reader to the beginning of its input stream. If the checkpoint is non-null, this method should position the reader to the last checkpointed position. This is a required method for an item reader.

Syntax:

```
@Open void <method-name>(Externalizable checkpoint) throws Exception
```

Where:

<method-name>	Is the name of the item reader Open method.

Example:

```
@ItemReader("MyItemReader")
public class MyItemReaderImpl {
	@Open void open(MyCheckpointData checkpoint) throws Exception {...}
}
```

5.1.1.1.2 @Close

The @Close annotation identifies a method that does cleanup processing for an item reader. This is an optional method for an item reader.

Syntax:

```
@Close void <method-name>() throws Exception
```

Where:

<method-name>	is the name of the item reader Close method.

Example:

```
@ItemReader("MyItemReader")
public class MyItemReaderImpl {
```

```
	@Close void close() throws Exception {...}
}
```

5.1.1.1.3 @ReadItem

The @ReadItem annotation identifies a method that reads the next item from an item reader. This method returns an item of the type defined by this reader. This is a required method for an item reader.

Syntax:

```
@ReadItem <item-type> <method-name> () throws Exception
```

Where:

<item-type>	is the item type returned by the item reader.
<method-name>	is the name of the item reader read item method.

Example:

```
@ItemReader("MyItemReader")
public class MyItemReaderImpl {
	@ReadItem MyBatchInputRecord read() throws Exception {...}
}
```

5.1.1.1.4 @GetCheckpointInfo

The @GetCheckpointInfo annotation identifies a method that receives control to provide the current checkpoint position of a reader. This method returns a Java Externalizable. This is a required method for an item reader.

Syntax:

```
@GetCheckpointInfo  Externalizable <method-name> () throws Exception
```

Where:

<method-name>	is the name of the item reader GetCheckpointInfo method.

Example:

```
@ItemReader("MyItemReader")
public class MyItemReaderImpl {
```

```
    @GetCheckpointInfo MyCheckpointData getChkpt() throws Exception {...}
}
```

5.1.1.2 @ItemProcessor

The @ItemProcess annotation identifies a class as an item processor, which is one of the three artifact types that comprise a chunk-type step.

Syntax:

@ItemProcessor[("<id>")]

Where:

<id>	is the optional name for the ItemProcessor. The default is the item processor class name.

Example:

```
@ItemProcessor("MyItemProcessor")
public class MyItemProcessorImpl {...}
```

5.1.1.2.1 @ProcessItem

The @ProcessItem annotation identifies a method that performs the business processing for an item processor. This method receives an input item for processing and returns an output item. The item processor defines both the input and output types. This is a required method for an item reader.

Syntax:

@ProcessItem <output-item-type> <method-name>(<item-type> item) throws Exception

Where:

<output-item-type>	is the Java type of the output item produced by the item processor.
<method-name>	is the name of the item process ProcessItem method.
<item-type>	is the Java type of the input item consumed by the item processor.

Example:

```
@ItemProcessor("MyItemProcessor")
public class MyItemProcessorImpl {
    @ProcessItem MyBatchOutputRecord process(MyBatchInputRecord item) throws Exception {...}
}
```

5.1.1.3 @ItemWriter

The @ItemWriter annotation identifies a class as an item writer, which is one of the three artifact types that comprise a chunk-type step.

Syntax:

@ItemWriter[([value=]"<id>"[,buffered=true|false])]

Where:

<id>	is the optional name for the item writer. The default is the item writer class name.
buffered	is an optional attribute that specifies whether writes done through this ItemWriter are buffered in chunks or written one item at a time.

Example:

```
@ItemWriter(value="MyItemWriter",buffered=false)
public class MyItemWriterImpl {...}
```

5.1.1.3.1 @Open

The @Open annotation identifies a method that does initialization processing for an item writer. This method receives a Java Externalizable that contains the last checkpoint position for the writer. If the checkpoint is null, this method should position the writer to the beginning of its output stream. If the checkpoint is non-null, this method should position the writer to the last checkpointed position. This is a required method for an item writer.

Syntax:

@Open void <method-name>(Externalizable checkpoint) throws Exception

Where:

<method-name>	Is the name of the item writer Open method.

Example:

```
@ItemWriter("MyItemWriter")
public class MyItemWriterImpl {
        @Open void open(MyCheckpointData checkpoint) throws Exception {...}
}
```

5.1.1.3.2 @Close

The @Close annotation identifies a method that does cleanup processing for an item writer. This is an optional method for an item writer.

Syntax:

```
@Close void <method-name>() throws Exception
```

Where:

<method-name>	is the name of the item writer Close method.

Example:

```
@ItemWriter("MyItemWriter")
public class MyItemWriterImpl {
	@Close void close() throws Exception {...}
}
```

5.1.1.3.3 @WriteItems

The @WriteItems annotation identifies a method that writes the next item for an item writer. This method receives an item to write to the writer's output stream. The method defines the item type. This is a required method for an item writer.

Syntax:

```
@WriteItems void <method-name> (List<item-type> items) throws Exception
```

Where:

<method-name>	is the name of the item writer WriteItem method.
List<item-type>	is a list of items of the type written by the item writer.

Example:

```
@ItemWriter("MyItemWriter")
public class MyItemWriterImpl {
	@WriteItems void write(MyBatchOutputRecord items) throws Exception {...}
}
```

5.1.1.3.4 @GetCheckpointInfo

The @GetCheckpointInfo annotation identifies a method that receives control to provide the current checkpoint position of a writer. This method returns a Java Externalizable. This is a required method for an item writer.

Syntax:

@GetCheckpointInfo Externalizable <method-name> () throws Exception

Where:

<method-name>	is the name of the item writer GetCheckpointInfo method.

Example:

```
@ItemWriter("MyItemWriter")
public class MyItemWriterImpl {
        @GetCheckpointInfo MyCheckpointData getChkpt() throws Exception {...}
}
```

5.1.1.4 @CheckpointAlgorithm

The @CheckpointAlgorithm annotation identifies a class as a custom checkpoint algorithm. A checkpoint algorithm is used to provide custom logic that decides when a checkpoint should occur during chunk processing.

Syntax:

@CheckpointAlgorithm [("<id>")]

Where:

<id>	is the optional name for the checkpoint algorithm. The default is the checkpoint algorithm class name.

```
@CheckpointAlgorithm("MyCheckpointAlgorithm")
public class MyCheckpointAlgorithmImpl{...}
```

5.1.1.4.1 @GetCheckpointTimeout

The @GetCheckpointTimeout annotation identifies a method that receives control at the beginning of a new checkpoint interval for the purpose of establishing the checkpoint transaction timeout. This method is invoked if and only if the chunk step to which it belongs is configured for global transactions. It is invoked before the next checkpoint transaction begins. It receives as input the value of the checkpoint-timeout attribute of the current chunk step. See section [REF] for more information about the checkpoint-timeout attribute. The @GetCheckpointTimeout method returns an integer value, which is the timeout value that will be used for the next checkpoint transaction. This method is useful to automate the setting of the checkpoint timeout based on factors known outside the job definition. This is an optional method for a checkpoint algorithm.

Syntax:

```
@GetCheckpointTimeout  int  <method-name> (int timeout) throws Exception
```

Where:

<method-name>	is the name of the GetCheckpointTimeout method.

Example:

```
@CheckpointAlgorithm("MyCheckpointAlgorithm")
public class MyCheckpointAlgorithmImpl {
	@GetCheckpointTimeout int getTimeout(int timeout) throws Exception {...}
}
```

5.1.1.4.2 @BeginCheckpoint

The @BeginCheckpoint annotation identifies a method that receives control at the beginning of a new checkpoint interval. It is invoked immediately after the next checkpoint transaction is started but before any items are processed. This is an optional method for a checkpoint algorithm.

Syntax:

```
@BeginCheckpoint  void <method-name> () throws Exception
```

Where:

<method-name>	is the name of the BeginCheckpoint method.

Example:

```
@CheckpointAlgorithm("MyCheckpointAlgorithm")
public class MyCheckpointAlgorithmImpl {
	@BeginCheckpoint void begin() throws Exception {...}
}
```

5.1.1.4.3 @IsReadyToCheckpoint

The @IsReadyToCheckpoint annotation identifies a method that decides whether or not to commit a checkpoint. It is called by the batch runtime if a custom checkpoint algorithm is configured on a chunk. This is a required method for a checkpoint algorithm. This method is processed after each call to the item processor. This is a required method for a checkpoint algorithm.

Syntax:

```
@IsReadyToCheckpoint  boolean  <method-name> () throws Exception
```

Where:

<method-name>	is the name of the IsReadyToCheckpoint method.

Example:

```
@CheckpointAlgorithm("MyCheckpointAlgorithm")
public class MyCheckpointAlgorithmImpl {
	@IsReadyToCheckpoint boolean ready() throws Exception {...}
}
```

5.1.1.4.4 @EndCheckpoint

The @EndCheckpoint annotation identifies a method that receives control at the beginning of a new checkpoint interval. It is invoked after all items in the current chunk have been processed by the item writer and immediately before the current checkpoint transaction is committed. This is an optional method for a checkpoint algorithm.

Syntax:

```
@EndCheckpoint  void <method-name> () throws Exception
```

Where:

<method-name>	is the name of the EndCheckpoint method.

Example:

```
@CheckpointAlgorithm("MyCheckpointAlgorithm")
public class MyCheckpointAlgorithmImpl {
        @EndCheckpoint void end() throws Exception {...}
}
```

5.1.2 @Batchlet

The @Batchlet annotation identifies a class as a Batchlet. A Batchlet-type step implements a roll your own batch pattern. This batch pattern is invoked once, runs to completion, and returns an exit status. This pattern must implement and honor a "cancel" callback to enable operational termination of the Batchlet.

Syntax:

```
@Batchlet[("<id>")]
```

Where:

<id>	is the optional name for the Batchlet. The default is the Batchlet class name.

Example:

```
@Batchlet("MyBatchLet")
public class MyBatchLet Impl{...}
```

5.1.2.1 @BeginStep

The @BeginStep annotation identifies a method that receives control at the beginning of Batchlet processing. This is an optional method for a Batchlet.

Syntax:

```
@BeginStep  void <method-name> () throws Exception
```

Where:

<method-name>	is the name of the Batchlet BeginStep method.

Example:

```
@Batchlet("MyBatchlet")
public class MyBatchletImpl {
        @BeginStep void begin() throws Exception {...}
}
```

5.1.2.2 @Process

The @Process annotation identifies a method that receives control to do the business processing for a Batchlet. This is a required method for a Batchlet. It must return a non-null, non-empty String that represents the user-defined exit status of the Batchlet.

Syntax:

```
@Process  String <method-name> () throws Exception
```

Where:

<method-name>	is the name of the Batchlet Process method.

Example:

```
@Batchlet("MyBatchlet")
public class MyBatchletImpl {
        @Process String process() throws Exception {...
                return "SUCCESS";
        }
}
```

5.1.2.3 @Cancel

The @Cancel annotation identifies a method that receives control in response to a cancel job operation while the the Batchlet is running. The BatchLet @Process method must periodically check if @Cancel has been invoked. If @Cancel has been invoked, @Process must return. This is a required method for a Batchlet.

Syntax:

```
@Cancel  void <method-name> () throws Exception
```

Where:

<method-name>	is the name of the Batchlet Cancel method.

Example:

```
@Batchlet("MyBatchlet")
public class MyBatchletImpl {
        @Cancel void cancel() throws Exception {...}
}
```

5.1.2.4 @EndStep

The @EndStep annotation identifies a method that receives control at the end of Batchlet processing. This is an optional method for a Batchlet.

Syntax:

```
@EndStep  void <method-name> () throws Exception
```

Where:

<method-name>	is the name of the Batchlet EndStep method.

Example:

```
@Batchlet("MyBatchlet")
public class MyBatchletImpl {
        @EndStep void end() throws Exception {...}
}
```

5.2 Listeners

Use Listeners to interpose on batch execution.

5.2.1 @JobListener

The @JobListener annotation identifies a class as a job listener. A job listener can receive control before and after a job execution runs.

Syntax:

```
@ JobListener [("<id>")]
```

Where:

<id>	is the optional name for the job listener. The default is the job listener class name.

Example:

```
@JobListener("MyJobListener")
public class MyJobListenerImpl {...}
```

5.2.1.1 @BeforeJob

The @BeforeJob annotation identifies a method that receives control before a job execution begins. This is an optional method for a job listener.

Syntax:

```
@BeforeJob  void <method-name> () throws Exception
```

Where:

<method-name>	is the name of the job listener BeforeJob method.

Example:

```
@JobListener("MyJobListener")
public class MyJobListenerImpl {
	@BeforeJob void before(){...}
}
```

5.2.1.2 @AfterJob

The @AfterJob annotation identifies a method that receives control after a job execution ends. This is an optional method for a job listener.

Syntax:

```
@AfterJob  void <method-name> () throws Exception
```

Where:

<method-name>	is the name of the job listener AfterJob method.

Example:

```
@JobListener("MyJobListener")
public class MyJobListenerImpl {
	@AfterJob void after(){…}
}
```

5.2.1.3 @BeforeStep

The @BeforeStep annotation identifies a method that receives control before a step execution begins. This is an optional method for a job listener. The @StepContext class field identifies the current step. See section [REF] for further information about @StepContext.

Syntax:

```
@BeforeStep  void <method-name> () throws Exception
```

Where:

<method-name>	is the name of the job listener BeforeStep method.

Example:

```
@JobListener("MyJobListener")
public class MyJobListenerImpl {
	@BeforeStep void before(){…}
}
```

5.2.1.4 @AfterStep

The @AfterStep annotation identifies a method that receives control after a step execution ends. This is an optional method for a job listener. The @StepContext class field identifies the current step. See section [REF] for further information about @StepContext.

Syntax:

```
@AfterStep  void <method-name> () throws Exception
```

Where:

<method-name>	is the name of the job listener AfterStep method.

Example:

```
@JobListener("MyJobListener")
public class MyJobListenerImpl {
	@AfterStep void after(){…}
}
```

5.2.2 @StepListener

The @StepListener annotation identifies a class as a step listener. A step listener can receive control before and after a step runs.

Syntax:

```
@ StepListener [("<id>")]
```

Where:

<id>	is the optional name for the step listener. The default is the step listener class name.

Example:

```
@StepListener("MyStepListener")
public class MyStepListenerImpl {...}
```

5.2.2.1 @BeforeStep

The @BeforeStep annotation identifies a method that receives control before a step execution begins. This is an optional method for a step listener.

Syntax:

```
@BeforeStep  void <method-name> () throws Exception
```

Where:

<method-name>	is the name of the step listener BeforeStep method.

Example:

```
@StepListener("MyStepListener")
public class MyStepListenerImpl {
        @BeforeStep void before(){…}
}
```

5.2.2.2 @AfterStep

The @AfterStep annotation identifies a method that receives control after a step execution ends. This is an optional method for a step listener.

Syntax:

@AfterStep void <method-name> () throws Exception

Where:

<method-name>	is the name of the step listener AfterStep method.

Example:

```
@StepListener("MyStepListener")
public class MyStepListenerImpl {
        @AfterStep void after(){…}
}
```

5.2.3 @ CheckpointListener

The @CheckpointListener annotation identifies a class as a checkpoint listener. A checkpoint listener can receive control before and after a checkpoint is taken.

Syntax:

@ CheckpointListener [("<id>")]

Where:

<id>	is the optional name for the checkpoint listener. The default is the checkpoint listener class name.

Example:

```
@CheckpointListener("MyCheckpointListener")
```

```
public class MyCheckpointListenerImpl {...}
```

5.2.3.1 @BeforeCheckpoint

The @BeforeCheckpoint annotation identifies a method that receives control before a checkpoint is taken. This is an optional method for a checkpoint listener.

Syntax:

```
@BeforeCheckpoint  void <method-name> () throws Exception
```

Where:

<method-name>	is the name of the checkpoint listener BeforeCheckpoint method.

Example:

```
@CheckpointListener("MyCheckpointListener")
public class MyCheckpointListenerImpl {
	@BeforeCheckpoint void before(){...}
}
```

5.2.3.2 @AfterCheckpoint

The @AfterCheckpoint annotation identifies a method that receives control after a checkpoint has been taken. This is an optional method for a checkpoint listener.

Syntax:

```
@AfterCheckpoint  void <method-name> () throws Exception
```

Where:

<method-name>	is the name of the checkpoint listener AfterCheckpoint method.

Example:

```
@CheckpointListener("MyCheckpointListener")
public class MyCheckpointListenerImpl {
	@AfterCheckpoint void after(){...}
}
```

5.2.4 @ItemReadListener

The @ItemReadListener annotation identifies a class as an item read listener. A item read listener can receive control before and after an item is read by an item reader, and also if the reader throws an exception.

Syntax:

@ ItemReadListener [("<id>")]

Where:

<id>	is the optional name for the item read listener. The default is the item read listener class name.

Example:

```
@ItemReadListener("MyItemReadListener")
public class MyItemReadListenerImpl {...}
```

5.2.4.1 @BeforeRead

The @BeforeRead annotation identifies a method that receives control before an item reader is called to read the next item. This is an optional method for an item read listener.

Syntax:

@BeforeRead void <method-name> () throws Exception

Where:

<method-name>	is the name of the item read listener BeforeRead method.

Example:

```
@ItemReadListener("MyItemReadListener")
public class MyItemReadListenerImpl {
        @BeforeRead void before(){…}
}
```

5.2.4.2 @AfterRead

The @AfterRead annotation identifies a method that receives control after an item reader reads an item. The method receives the item read as an input. This is an optional method for an item read listener.

Syntax:

@AfterRead void <method-name> (<item-type> item) throws Exception

Where:

<method-name>	is the name of the item read listener AfterRead method.
<item-type>	is the item type read by the item reader.

Example:

```
@ItemReadListener("MyItemReadListener")
public class MyItemReadListenerImpl {
        @AfterRead void write(MyBatchOutputRecord item) throws Exception {...}
}
```

5.2.4.3 @OnReadError

The @OnReadError annotation identifies a method that receives control after an item reader throws an exception. This method receives the exception as an input. This is an optional method for an item read listener.

Syntax:

@OnReadError void <method-name> (Exception ex) throws Exception

Where:

<method-name>	is the name of the item read listener OnReadError method.

Example:

```
@ItemReadListener("MyItemReadListener")
public class MyItemReadListenerImpl {
        @OnReadError void onError(Exception ex) throws Exception {...}
}
```

5.2.5 @ItemProcessListener

The @ItemProcessListener annotation identifies a class as an item processor listener. An item processor listener can receive control before and after an item is processed by an item processor, and also if the processor throws an exception.

Syntax:

@ ItemProcessListener [("<id>")]

Where:

<id>	is the optional name for the item processor listener. The default is the item processor listener class name.

Example:

```
@ItemProcessListener("MyItemProcessListener")
public class MyItemProcessListenerImpl {...}
```

5.2.5.1 @BeforeProcess

The @BeforeProcess annotation identifies a method that receives control before an item processor is called to process the next item. The method receives the item to be processed as an input. This is an optional method for an item processor listener.

Syntax:

@BeforeProcess void <method-name> (<item-type> item) throws Exception

Where:

<method-name>	is the name of the item processor listener BeforeProcess method.
<item-type>	is the input item type processed by the item processor.

Example:

```
@ItemProcessListener("MyItemProcessListener")
public class MyItemProcessListenerImpl {
```

```
    @BeforeProcess void before(MyBatchInputRecord item){…}
}
```

5.2.5.2 @AfterProcess

The @AfterProcess annotation identifies a method that receives control after an item processor processes an item. The method receives the item processed and the result item as an input. This is an optional method for an item processor listener.

Syntax:

@AfterProcessor void <method-name> (<item-type> item, <result-type> result) throws Exception

Where:

<method-name>	is the name of the item processor listener AfterProcess method.
<item-type>	is the input item type processed by the item processor.
<result-type>	is the output type produced by the item processor.

Example:

```
@ItemProcessListener("MyItemProcessListener")
public class MyItemProcessListenerImpl {
    @AfterProcessor void write(MyBatchInputRecord item, MyBatchOutputRecord result) throws
Exception {...}
}
```

5.2.5.3 @OnProcessError

The @OnProcessError annotation identifies a method that receives control after an item processor throws an exception. This method receives the exception and the input item. This is an optional method for an item processor listener.

Syntax:

@OnProcessError void <method-name> (Exception ex, <item-type> item) throws Exception

Where:

<method-name>	is the name of the item processor listener OnProcessError method.
<item-type>	is the input item type processed by the item processor.

Example:

```
@ItemProcessListener("MyItemProcessListener")
public class MyItemProcessListenerImpl {
	@OnProcessError void onError(Exception ex, MyBatchInputRecord item) throws Exception {...}
}
```

5.2.6 @ItemWriteListener

The @ItemWriteListener annotation identifies a class as an item write listener. A item write listener can receive control before and after an item is written by an item writer, and also if the writer throws an exception.

Syntax:

@ ItemWriteListener [("<id>")]

Where:

<id>	is the optional name for the item write listener. The default is the item write listener class name.

Example:

```
@ItemWriteListener("MyItemWriteListener")
public class MyItemWriteListenerImpl {...}
```

5.2.6.1 @BeforeWrite

The @BeforeWrite annotation identifies a method that receives control before an item writer is called to write its items. The method receives the list of items sent to the item reader as an input. This is an optional method for an item write listener.

Syntax:

@BeforeWrite void <method-name> (List<item-type> items) throws Exception

Where:

<method-name>	is the name of the item write listener BeforeWrite method.
List<item-type>	is the list of items sent to the item writer.

Example:

```
@ItemWriteListener("MyItemWriteListener")
public class MyItemWriteListenerImpl {
	@BeforeWrite void before(List<MyBatchOutputRecord> items) throws Exception {...}
}
```

5.2.6.2 @AfterWrite

The @AfterWrite annotation identifies a method that receives control after an item writer writes its items. The method receives the list of items sent to the item reader as an input. This is an optional method for an item write listener.

Syntax:

```
@AfterWrite void <method-name> (List<item-type> items) throws Exception
```

Where:

<method-name>	is the name of the item write listener AfterWrite method.
List<item-type>	is the list of items sent to the item writer.

Example:

```
@ItemWriteListener("MyItemWriteListener")
public class MyItemWriteListenerImpl {
	@AfterWrite void write(List<MyBatchOutputRecord> items) throws Exception {...}
}
```

5.2.6.3 @OnWriteError

The @OnWriteError annotation identifies a method that receives control after an item writer throws an exception. The method receives the list of items sent to the item reader as an input. This is an optional method for an item write listener.

Syntax:

```
@OnWriteError void <method-name> (Exception ex, List<item-type items) throws Exception
```

Where:

<method-name>	is the name of the item write listener OnWriteError method.

List<item-type>	is the list of items sent to the item writer.

Example:

```
@ItemWriteListener("MyItemWriteListener")
public class MyItemWriteListenerImpl {
        @OnWriteError void onError(Exception ex, List<MyBatchOutputRecord> items) throws
Exception {...}
}
```

5.2.7 @SkipListener

The @SkipListener annotation identifies a class as a skip listener. A skip listener can receive control when a skippable exception is thrown from an item reader, processor, or writer.

Syntax:

```
@SkipListener [("<id>")]
```

Where:

<id>	is the optional name for the skip listener. The default is the skip listener class name.

Example:

```
@SkipListener("MySkipListener")
public class MySkipListenerImpl {...}
```

5.2.7.1 @OnSkipInRead

The @OnSkipInRead annotation identifies a method that receives control when a skippable exception is thrown from an item reader. The method receives the exception as an input. This is an optional method for a skip listener.

Syntax:

```
@OnSkipInRead void onSkipInRead(Exception ex) {…}
```

Where:

<method-name>	is the name of the skip listener OnSkipInRead method.

Example:

```
@SkipListener("SkipListener")
public class SkipListenerImpl {
	@OnSkipInRead void onSkip(Exception ex) throws Exception {...}
}
```

5.2.7.2 @OnSkipInProcess

The @OnSkipInProcess annotation identifies a method that receives control when a skippable exception is thrown from an item processor. The method receives the exception and the item to process as an input. This is an optional method for a skip listener.

Syntax:

```
@OnSkipInProcess void <method-name>(Exception ex, <item-type> item) {...}
```

Where:

<method-name>	is the name of the skip listener OnSkipInRead method.
<item-type>	is the type of item received by the item processor.

Example:

```
@SkipListener("SkipListener")
public class SkipListenerImpl {
	@OnSkipInProcess void onSkip(Exception ex, MyBatchInputRecord item) throws Exception {...}
}
```

5.2.7.3 @OnSkipInWrite

The @OnSkipInWrite annotation identifies a method that receives control when a skippable exception is thrown from an item writer. The method receives the exception and the item that was skipped during write as an input. This is an optional method for a skip listener.
Syntax:

```
@OnSkipInProcess void <method-name>(Exception ex, <item-type> item) {...}
```

Where:

<method-name>	is the name of the skip listener OnSkipInRead method.
<item-type>	is the type of item received by the item writer.

Example:

```
@SkipListener("SkipListener")
public class SkipListenerImpl {
	@OnSkipInWrite void onSkip(Exception ex, MyBatchInputRecord item) throws Exception {...}
}
```

5.2.8 @RetryListener

The @RetryListener annotation identifies a class as a retry listener. A retry listener can receive control when a retryable exception is thrown from a chunk.

Syntax:

```
@RetryListener [("<id>")]
```

Where:

<id>	is the optional name for the retry listener. The default is the retry listener class name.

Example:

```
@RetryListener ("MyRetryListener ")
public class MyRetryListener Impl {...}
```

5.2.8.1 @OnRetryException

The @OnRetryException annotation identifies a method that receives control when a retryable exception is thrown from an item writer. The method receives as input the exception and the item that was being written. The method receives control in same transactional scope as the item writer. If this method throws a an exception, the current transaction is rolled back to the last checkpoint and the entire chunk is retried. This is true even if the original exception or the exception thrown by this

method is specified as non-rollback. If this method is defined, it receives control before the @OnRetryItem method (if defined). See next section for details on @OnRetryItem method. This is an optional method for a retry listener.

Where:

<method-name>	is the name of the retry listener OnRetryException method.
<item-type>	is the type of item received by the item writer.

Example:

```
@RetryListener ("MyRetryListener ")
public class MyRetryListener Impl {
	@OnRetryException void retryException(Exception ex, MyBatchInputRecord item) throws
Exception {...}
}
```

5.2.8.2 @OnRetryItem

The @OnRetryItem annotation identifies a method that receives control when a retryable exception is thrown from an item writer. The method receives as input the exception and the item that was being written. If the exception is a non-rollback exception, this method receives control in the same transaction scope as the item writer. If this is a rollback exception (or @OnRetryException throws an exception - see section [REF]), the following occurs in this sequence:

6. the current transaction is rolled back;
7. a new transaction scope started;
8. the items up to, but not including, this item are re-processed by the item writer;
9. the current item is processed through this method (@OnRetryItem);
10. the current item is then processed through the item writer;
11. the remaining items in the current chunk are processed through the item writer.

If this method throws a skippable exception, the item is processed according to item writer skip processing rules. If this method throws a non-skippable exception, the current transaction is rolled back, and the step ends with a batch status of FAILED. If defined, the @OnRetryItem method receives control after the @OnRetryException method (if defined).   See section [REF] for details on the

@OnRetryException method. This is an optional method for a retry listener.

Syntax:

@OnRetryItem void <method-name>(Exception ex, <item-type> item) throws Exception

Where:

<method-name>	is the name of the retry listener OnRetryItem method.
<item-type>	is the type of item received by the item writer.

Example:

```
@RetryListener ("MyRetryListener ")
public class MyRetryListener Impl {
        @OnRetryItem void retryItem(Exception ex, MyBatchInputRecord item) throws Exception {...}
}
```

5.3 Batch Properties

Batch applications need a way to receive parameters when a job is initiated for execution. Properties can be defined by batch programming model artifacts, then have values passed to them when a job is initiated. Batch properties are string values.

5.3.1 @BatchProperty

The @BatchProperty annotation identifies a class field as a batch property. A batch property has a name (name) and default value. The @BatchProperty may be used on a class field for any class identified as a batch programming model artifact - e.g. ItemReader, ItemProcessor, JobListener, etc..

Syntax:

@BatchProperty(name="<property-name>"[, value="<default-value"]) String <field-name>;

Where:

<property-name>	is the name of this batch property.

<default-value>	is the default String value for this batch property. If not specified, the default is an empty string.
<field-name>	is the field name of the batch property.

Example:

```
@ItemReader("MyItemReader")
public class MyItemReaderImpl {
	@BatchProperty(name="fileName","/tmp/input.txt") String fname;
}
```

Behavior:

When the batch runtime instantiates the batch artifact (item reader in this example), it assigns the value of the named property provided on the job initiation to the corresponding @BatchProperty field. If no value is provided in the job initiation, the batch runtime assigns the default from the @BatchProperty annotation to the field.

5.4 Batch Contexts

Context objects are supplied by the batch runtime and provide important functions to a batch application. Contexts provide information about the running batch job, provide a place for a batch job to store interim values, and provide a way for the batch application to communicate important information back to the batch runtime. Contexts can be injected into an application as member variables. There is a context for both job and step. The job context represents the entire job. The step context represents the current step executing within the job.

5.4.1 @BatchContext

The @BatchContext annotation identifies a class field as a batch context. Such a class field is initialized with a context object according to type by the batch runtime. Batch contexts are runtime objects available during job execution. They provide a communication mechanism between the runtime and the batch artifacts of a batch application.

Syntax:

@BatchContext <context-type> <field-name>;

Where:

<context-type>	is the context type. Supported types are: JobContext , StepContext, FlowContext,

	and SplitContext
<field-name>	is the field name of the context.

Example:

```
@JobListener("MyJobListener")
public class MyJobListenerImpl {
        @BatchContext  JobContext  jctxt;
}
```

See section [REF] for definition of JobContext class.

```
@StepListener("MyStepListener")
public class MyStepListenerImpl {
        @BatchContext  StepContext  sctxt;
}
```

See section [REF] for definition of StepContext class.

5.4.1.1 Batch Context Lifecycle and Scope

A batch context has thread affinity and is visible only to the batch artifacts executing on that particular thread. A @BatchContext annotated field may be null when out of scope. Each context type has a distinct scope and lifecycle as follows:

1. JobContext

   There is one JobContext per job execution. It exists for the life of a job. There is a distinct JobContext for each sub-thread of a parallel execution (e.g. partitioned step).

2. StepContext

   There is one StepContext per step execution. It exists for the life of the step. For a partitioned step, there is one StepContext for the parent step/thread; there is a distinct StepContext for each sub-thread.

3. FlowContext

   There is one FlowContext per flow execution. It exists for the life of the flow. For a partitioned flow, there is one FlowContext for the parent flow/thread; there is a distinct FlowContext for each sub-thread.

4. SplitContext

   There is one SplitContext per split execution. It exists for the life of the split. There is one SplitContext for the parent split/thread; there is a distinct SplitContext for each sub-thread of a split.

5.4.1.2 Batch Context and @Decider

The @Decider batch artifact receives control between execution elements: between a step, split, or flow. When it receives control, only the batch context corresponding to the last execution element to run can be in scope.

Example:

```
@Decider("MyDecider")
public class MyDeciderImpl {
	@BatchContext JobContext jobCtxt= null;
	@BatchContext StepContext stepCtxt= null;
	@BatchContext FlowContext flowCtxt= null;
	@BatchContext SplitContext splitCtxt= null;
	if (jobCtxt == null) System.out.println("This can never happen!");
	if (stepCtxt != null) System.out.println("Decider  preceded by a step.");
	if (flowCtxt != null) System.out.println("Decider  preceded by a flow.");
	if (splitCtxt != null) System.out.println("Decider  preceded by a split.");
}
```

5.5 Parallelization

Batch jobs may be configured to run some of their steps in parallel. There are two supported parallelization models:

1. partitioned

   In the partitioned model, a step or flow is configured to run as multiple instances across multiple threads. Each thread runs the same step or flow. This model is logically equivalent to launching multiple instances the same step or flow. It is intended that each partition processes a different range of the input items.

2. concurrent

   In the concurrent model, the flows defined by a split are configured to run concurrently on

multiple threads, one flow per thread.

5.5.1 @PartitionAlgorithm

The @PartitionAlgorithm annotation identifies a class as a partition algorithm. A partition algorithm receives control at the start of a partitioned execution. The partition algorithm is responsible to establish the number of partitions for a partitioned step or flow. It is also responsible to provide unique batch properties to serve as parameters to each partition.

Syntax:

@PartitionAlgorithm [("<id>")]

Where:

<id>	is the optional name for the partition algorithm. The default is the partition algorithm class name.

Example:

```
@PartitionAlgorithm("MyPartitionAlgorithm")
public class MyPartitionAlgorithmImpl { }
```

5.5.1.1 @CalculatePartitions

The @CalculatePartitions annotation identifies a method that receives control at the start of parallel processing for a partitioned step or flow. The method must return a PartitionPlan, which instructs the runtime how many partitions to execute and provides batch properties for each partition. This is a required method for a partition algorithm.
Syntax:

@CalculatePartitions PartitionPlan <method-name>( ) throws Exception

Where:

<method-name>	is the name of the partition algorithm CalculatePartitions method.

Example:

```
@CalculatePartitions ("CalculatePartitions ")
public class MyPartitionAlgorithmImpl {
```

```
        @CalculatePartitions PartitionPlan calculatePartitions () throws Exception {...}
}
```

See section [REF] for details on the PartitionPlan result value type.

Implementation notes: ensure jobname and txid in job context

5.5.2 @LogicalTX

The @LogicalTX annotation identifies a class as a logical transaction synchronization for parallel processing. A logical transaction provides a unit of work demarcation across parallel threads. It is not a JTA transaction; no resources are enlisted. Rather, it provides transactional flow semantics to facilitate carrying out of compensation logic.

Syntax:

```
@LogicalTX [("<id>")]
```

Where:

<id>	is the optional name for the logical transaction synchronization. The default is the logical transaction synchronization class name.

Example:

```
@LogicalTX("MyLogicalTX ")
public class MyLogicalTX Impl { }
```

5.5.2.1 @LogicalTXBegin

The @LogicalTXBegin annotation identifies a method that receives control at the start of a logical transaction for parallel processing. It receives control before the partition algorithm is invoked and before any sub-jobs are started. This is an optional method for a logical transaction synchronization.

Syntax:

```
@LogicalTXBegin void <method-name>()  throws Exception
```

Where:

<method-name>	is the name of the logical synchronization LogicalTXBegin method.

Example:

```
@LogicalTX ("MyLogicalTX ")
public class MyLogicalTX Impl {
	@LogicalTXBegin void begin()  throws Exception {...}
}
```

5.5.2.2 @LogicalTXBeforeCompletion

The @LogicalTXBeforeCompletion annotation identifies a method that receives control at the start of a parallel processing . It receives control after all sub-jobs have completed. It does not receive control if the logical transaction is rolling back. This is an optional method for a logical transaction synchronization.

Syntax:

```
@LogicalTXBeforeCompletion void <method-name>()  throws Exception
```

Where:

<method-name>	is the name of the logical synchronization LogicalTXBeforeCompletion method.

Example:

```
@LogicalTX ("MyLogicalTX ")
public class MyLogicalTX Impl {
	@LogicalTXBeforeCompletion void beforeCompletion()  throws Exception {...}
}
```

5.5.2.3 @LogicalTXRollback

The @LogicalTXRollback annotation identifies a method that receives control if the runtime is rolling back a logical transaction. Any sub-jobs still running are stopped before this method is invoked. This method receives control if any of the following conditions are true:

1. One or more sub-jobs end with a Batch Status of STOPPED or FAILED.
2. Any of the following parallel callback objects throw an exception during the life of a logical transaction.
    a. PartitionAlgorithm

    b. Synchronization
    c. SubJobCollector
    d. SubJobAnalyzer
3. A parallel job is restarted and must perform logical transaction recovery. Logical transaction recovery is necessary when a parallel job terminates unexpectedly (e.g. JVM failure) and a logical transaction is left behind uncompleted.

This is an optional method for a logical transaction synchronization.

Syntax:

```
@LogicalTXRollback void <method-name>()  throws Exception
```

Where:

<method-name>	is the name of the logical synchronization LogicalTXRollback method.

Example:

```
@LogicalTX ("MyLogicalTX ")
public class MyLogicalTX Impl {
        @LogicalTXRollback void rollBack()  throws Exception {...}
}
```

5.5.2.4 @LogicalTXAfterCompletion

The @LogicalTXAfterCompletion annotation identifies a method that receives control at the end of a logical transaction. It receives a status string that identifies the outcome of the logical transaction. The status string value is either "COMIT" or "ROLLBACK". This is an optional method for a logical transaction synchronization.

Syntax:

```
@LogicalTXAfterCompletion void <method-name>(String status)  throws Exception
```

Where:

<method-name>	is the name of the logical synchronization LogicalTXAfterCompletion method.

Example:

```
@LogicalTX ("MyLogicalTX ")
public class MyLogicalTX Impl {
        @LogicalTXAfterCompletion void afterCompletion(String status)  throws Exception {...}
}
```

5.5.3 @SubJobCollector

The purpose of the sub-job collector is to provide a means for sharing transient data from sub-jobs to a single point of control belonging to their parent job. The SubJobAnalyzer is used to receive and process this data. See section [REF] for further information about the SubJobAnalyzer.

The @SubJobCollector annotation identifies a class as a data collector for parallel sub-jobs. A sub-job collector provides a means during parallel processing for sub-jobs to communicate data to their parent jobs. This is useful to contribute transient interim results from the sub-job to the parent job. The sub-job collector is invoked after each checkpoint and once more at the end of the step for a chunking step and only at the end of step for a Batchlet step.

Syntax:

```
@SubJobCollector [("<id>")]
```

Where:

<id>	is the optional name for the sub-job collector. The default is the sub-job collector class name.

Example:

```
@SubJobCollector ("MySubJobCollector")
public class MySubJobCollector Impl { }
```

5.5.3.1 @CollectSubJobData

The @CollectSubJobData annotation identifies a method that receives control at the end of a logical transaction. It receives a status string that identifies the outcome of the logical transaction. The status string value is either "COMIT" or "ROLLBACK". This is an optional method for a logical transaction synchronization.

Syntax:

@CollectSubJobData Externalizable <method-name>() throws Exception

Where:

<method-name>	is the name of the sub-job collector CollectSubJobData method.

Example:

```
@SubJobCollector ("MySubJobCollector")
public class MySubJobCollector Impl {
	@CollectSubJobData void collect()  throws Exception {...}
}
```

5.5.4 @SubJobAnalyzer

The @SubJobAnalyzer annotation identifies a class as a sub-job analyzer for a parallel job. A sub-job analyzer is invoked and receives the payload from the sub-job collector (if configured) each time the sub-job collector sends one out. The sub-job analyzer is also invoked and receives the sub-job exit status each time a sub-job ends.   The sub-job analyzer may be used to provide the final exit status for the parallel job.

Syntax:

@SubJobAnalyzer [("<id>")]

Where:

<id>	is the optional name for the sub-job analyzer. The default is the sub-job analyzer class name.

Example:

```
@SubJobAnalyzer ("MySubJobAnalyzer")
public class MySubJobAnalyzerImpl { }
```

5.5.4.1 @AnalyzeCollectorData

The @AnalyzeCollectorData annotation identifies a method that receives control each time a SubJob collector sends its payload. It receives as an input the Externalizable object from the collector. This is an optional method for a logical transaction synchronization.

Syntax:

@AnalyzeCollectorData void <method-name>(Externalizable data) throws Exception

Where:

<method-name>	is the name of the sub-job analyzer AnalyzeCollectorData method.

Example:

```
@SubJobAnalyzer ("MySubJobAnalyzer")
public class MySubJobAnalyzerImpl {
        @AnalyzeCollectorData void analyze(Externalizable data)  throws Exception {...}
}
```

5.5.4.2 @AnalyzeExitStatus

The @AnalyzeExitStatus annotation identifies a method that receives control each time a partition or split subjob ends. It receives as input the exit status string of the subjob. This is an optional method for a SubJob Analyzer.

Syntax:

@AnalyzeExitStatus void <method-name>(String exitStatus) throws Exception

Where:

<method-name>	is the name of the sub-job analyzer AnalyzeExitStatus method.

Example:

```
@SubJobAnalyzer ("MySubJobAnalyzer")
public class MySubJobAnalyzerImpl {
```

```
	@AnalyzeExitStatus void analyze(String exitStatus)  throws Exception {...}
}
```

5.6 @Decider

The @Decider annotation identifies a class as a decider. A decider may be used to determine batch exit status and sequencing between steps, splits, and flows in a Job XML. The decider returns a String value which becomes the exit status value on which the decision chooses the next transition.

Syntax:

```
@Decider [("<id>")]
```

Where:

<id>	is the optional name for the decider. The default is the decider class name.

Example:

```
@Decider ("MyDecider")
public class MyDeciderImpl { }
```

5.6.1 @Decide

The @Decide annotation identifies a method that receives control during job processing to set exit status between a step, split, or flow.

Syntax:

```
@Decide String <method-name>()  throws Exception
```

Where:

<method-name>	is the name of the Decider Decide method.

Example:

```
@Decider ("MyDecider")
```

```
public class MyDeciderImpl {
	@Decide String setExitStatus() throws Exception {
		return "EARLY COMPLETION";
	}
}
```

5.7 Batch Artifact Loader

A Batch Artifact Loader is responsible to load and instantiate batch artifact classes. Batch artifacts are specified by name in Job XML via the "ref" attribute. The batch runtime supplies a built-in Batch Artifact Loader that operates off batch.xml (see section [REF]. A batch application may supply a preferred Batch Artifact Loader that gets the first opportunity to load a batch artifact. The built-in loader is always the loader of last resort: if the preferred loader is configured, but cannot load a batch artifact, the built-in loader is given an opportunity to do so.

5.7.1 Batch Artifact Loader API

An application that supplies a preferred batch artifact loader must provide an implementation of the Batch Artifact Loader API. Batch artifacts are identified by the value of the "ref" attribute from the various elements of the Job XML model. A Batch Artifact Loader is required to return an object corresponding to the input "ref" value. If it is unable to load the referenced object, it must return null.

Interface:

```
package javax.batch.artifact.loader;

public interface ArtifactLoader {

	/**
	 * The load method loads an object corresponding
	 * to a ref value from a Job XML.
	 * @param ref value from Job XML
	 * @return object corresponding to ref value
	 * @throws Exception if artifact cannot be loaded.
	 */
	public Object load(String ref) throws Exception;

}
```

Notes:

1. The loader must supply a public default constructor.

5.7.2 batch.xml

The optional batch.xml file is used for two purposes:

1. to specify a preferred batch artifact loader (see section [REF)
2. to specify "ref-to-class" mappings for the default batch artifact loader
3. The preferred batch artifact loader is defined in batch.xml.

Syntax:

<loader classname="{*class name*}"/>

<artifact id="{*ref id*}" classname="${*class name*}"/>

Where:

loader classname	Specifies the fully-qualified class name of the preferred artifact loader.
artifact id	Specifies the "ref" id of a batch artifact. This value correspondes to the value on a "ref" attribute in Job XML.
artifact classname	Specifies the fully-qualified class name of the implementation corresponding to the artifact id.

Example:

```
<loader  classname="com.ibm.batch.sample.artifact.ArtifactLoaderImpl"/>
<artifact id="MyReader" classname="com.ibm.batch.sample.MyReaderImpl"/>
<artifact id="MyWriter" classname="com.ibm.batch.sample.MyWriterImpl"/>
```

6 Batch Runtime Specification

6.1 Job Identifiers

Jobs have names that are uniquely identified by the batch runtime according to the following naming convention:

{Jobid}:{Instanceid} :{ Executionid}

Where:

Jobid	is the value from the "id" attribute of the Job XML "job" element. e.g. <job id="Job1">
Instanceid	is a long that represents the next instance of a job id. A new job instance is created everytime a new job is started with the JobOperator "start" method.
Executionid	Is a long that represents the next attempt to run a particular job instance. A new execution is created everytime an existing job instance is restarted with the JobOperator "restart" method.

Each time a job id is started, a new instance is created. Each instance has one or more executions. When an instance is first created, its first execution is created. Each time an instance is restarted, a new execution is created.

6.2 JobOperator

The JobOperator interface provides a set of operations that act on

6.3 Classloader Scope

The batch runtime and the batch artifacts it loads execute in the class loader scope of the their initiator. This means if a typical Java main program starts a job through the JobOperator interface, the batch runtime and the batch artifacts it loads are loaded by the JVM system class loader. If a Java EE application starts a job, the Java EE application class loader hiearchy does the loading. This same principle applies to other arrangements, in which the job initiating environment supplies its own unique class loader configuration.

6.4 Job Path

The job path is the set of directories from which job XML is read for the purpose of starting a job. The default job path is the current directory. The "javax.jobpath" system property may be set to specify a list of directories for the job path. All files that end with ".xml" found on jobpath are read when the batch runtime initializes. Each time the JobOperator start method is invoked, jobpath is searched for

new files. The "id=" values from the top-level job, step, flow, and split elements are read and kept in memory. These values are used when the jobOperator start method is invoked to identify the target job, and also to indentify parent elements to resolve Job XML inheritance (see section [REF]).

Syntax:

javax.jobpath=${*directory-spec*}

Where:

javax.jobpath	specifies a list of directory identifies, separated by the platform directory separator character.

Examples:

Linux java command line:

-Djavax.jobpath="./:/tmp/jobs:/usr/bin/jobs"

6.5 Packaging Model

The batch artifacts that comprise a batch application requiring no unique packaging. They may be packaged in a standard jar file or can be included inside any Java archive type, as supported by the target execution platform in question. E.g. batch artifacts may be included in wars, EJB jars, etc, so long as they exist in the class loader scope of the program initiating the batch jobs (i.e. using the JobOperator start method).

The optional batch.xml file exists under a META-INF directory anywhere on the effective classpath.

6.6 Java EE Environment

6.6.1 Transaction Handling

In a Java EE environment, checkpoints must be managed in the context of a global transaction, although a compliant implementation may allow this as an optional behavior.

6.6.2 Batch Artifact Loading

The Java EE environment uses CDI to load batch artifacts. The batch.xml file is unnecessary in the Java EE environment unless a batch artifact loading mechanism other than CDI is desired - e.g. Spring DI.

6.6.3 JNDI

The JobOperator interface is exposed in the Java EE environment in the local JNDI namespace as:

- java:comp/JobOperator

6.7 Java SE Environment

6.7.1 Factory Method

To start the batch runtime in the Java SE environment, use the factory method:
import javax.batch.runtime.BatchRuntime;
javax.batch.operations.JobOperator;
BatchRuntime batchrt= BatchRuntime.getRuntime();
JobOperator jobop= batchrt.getJobOperator();

6.7.2 Transaction Handling

In a Java SE environment, checkpoints must be managed without a global transaction.

6.7.3 Batch Artifact Loading

In the Java SE environment, the batch.xml file is required for batch artifact loading. The batch.xml file must contain either:

1. <artifact> elements in the batch.xml to exploit the built-in batch artifact loader;
2. <loader> element specifying a preferred batch artifact loader.

For example, a preferred batch artifact loader could be supplied that exploited the "Weld" CDI reference implementation for batch artifact loading.

6.8 Supporting Classes

6.8.1 JobContext

Note to author: add parent job id?

```
package javax.batch.runtime.context;
/**
  *
  * JobContext is the class field type associated with the @JobContext
  * annotation. A JobContext provides information about the current
  * job execution.
  *
  * @see javax.batch.annotation.context.JobContext
  */
import java.util.Properties;
import javax.batch.runtime.metric.Metric;

public interface JobContext <T> {
      /**
       * The getId method returns the current job's identity.
       * @return job id string
       */
      public String getId();
      /**
       * The getProperties method returns the job level properties
       * specified in a job definition.
       * @return job level properties
       */
      public Properties getProperties();
      /**
       * The getTransientUserData method returns a transient data object
       * belonging to the current job. This data is destroyed after the
       * job ends.
       * @return user-specified type
       */
      public T getTransientUserData();
      /**
       * The setTransientUserData method stores a transient data object into
       * the current JobContext. This data is destroyed after the job ends.
       * @param data is the user-specified type
       */
      public void setTransientUserData(T data);
      /**
       * The getBatchStatus method returns the current batch status of the
       * current job. This value is set by the batch runtime and changes as
       * the batch status changes.
```

```
     * @return batch status string
     */
    public String getBatchStatus();
    /**
     * The getExitStatus method simply returns the exit status value stored
     * into the job context through the setExitStatus method or null.
     * @return exit status string
     */
    public String getExitStatus();
    /**
     * The setExitStatus method assigns the user-specified exit status for
     * the current job. When the job ends, the exit status of the job is
     * the value specified through setExitStatus. If setExitStatus was not
     * called or was called with a null value, then the exit status
     * defaults to the batch status of the job.
     * @Param status string
     */
    public void setExitStatus(String status);
    /**
     * The getMetrics method returns an array of job level metrics. This
     * includes elapsed time.
     * @see javax.batch.runtime.metric.Metric for definition of standard
     * metrics.
     * @return metrics array
     */
    public Metric[] getMetrics();
}
```

6.8.2 StepContext

Note to author: add start attempts?

```
package javax.batch.runtime.context;
/**
 *
 * StepContext is the class field type associated with the @StepContext
 * annotation. A StepContext provides information about the current step
 * of a job execution.
 *
 * @see javax.batch.annotation.context.StepContext
 */
import java.util.Properties;
import javax.batch.runtime.metric.Metric;

public interface StepContext <T,P extends Externalizable> {
    /**
     * The getId method returns the current step's identity.
     * @return step id string
     */
    public String getId();
    /**
     * The getProperties method returns the step level properties
     * specified in a job definition.
```

```
 * @return job level properties
 */
public Properties getProperties();
/**
 * The getTransientUserData method returns a transient data object
 * belonging to the current step. This data is destroyed after the
 * step ends.
 * @return user-specified type
 */
public T getTransientUserData();
/**
 * The setTransientUserData method stores a transient data object into
 * the current StepContext. This data is destroyed after the step ends.
 * @param data is the user-specified type
 */
public void setTransientUserData(T data);
/**
 * The getPersistentUserData method returns a persistent data object
 * belonging to the current step. The user data type must implement
 * java.util.Externalizable. This data is saved as part of a step's
 * checkpoint. For a step that does not do checkpoints, it is saved
 * after the step ends. It is available upon restart.
 * @return user-specified type
 */
public P getPersistentUserData();
/**
 * The setPersistentUserData method stores a persistent data object
 * into the current step. The user data type must implement
 * java.util.Externalizable. This data is saved as part of a step's
 * checkpoint. For a step that does not do checkpoints, it is saved
 * after the step ends. It is available upon restart.
 * @param data is the user-specified type
 */
public void setPersistentUserData(P data);
/**
 * The getBatchStatus method returns the current batch status of the
 * current step. This value is set by the batch runtime and changes as
 * the batch status changes.
 * @return batch status string
 */
public String getBatchStatus();
/**
 * The getExitStatus method simply returns the exit status value stored
 * into the step context through the setExitStatus method or null.
 * @return exit status string
 */
public String getExitStatus();
/**
 * The setExitStatus method assigns the user-specified exit status for
 * the current step. When the step ends, the exit status of the step is
 * the value specified through setExitStatus. If setExitStatus was not
 * called or was called with a null value, then the exit status
 * defaults to the batch status of the step.
 * @Param status string
 */
public void setExitStatus(String status);
/**
```

```java
         * The getException method returns the last exception thrown from a
         * step level batch artifact to the batch runtime.
         * @return the last exception
         */
        public Exception getException();
        /**
         * The getMetrics method returns an array of step level metrics. These
         * are things like commits, skips, etc.
         * @see javax.batch.runtime.metric.Metric for definition of standard
         * metrics.
         * @return metrics array
         */
        public Metric[] getMetrics();
}
```

6.8.3 FlowContext

```java
package javax.batch.runtime.context;
/**
  *
  * FlowContext is a class field type associated with the @BatchContext
  * annotation. A FlowContext provides information about the current
  * Flow execution.
  *
  * @see javax.batch.annotation.context.BatchContext
  */

public interface FlowContext {
        /**
         * The getId method returns the current flow's identity.
         * @return flow id string
         */
        public String getId();
        /**
         * The getBatchStatus method returns the current batch status of the
         * current job. This value is set by the batch runtime and changes as
         * the batch status changes.
         * @return batch status string
         */
        public String getBatchStatus();
        /**
         * The getExitStatus method simply returns the exit status value stored
         * into the flow context through the setExitStatus method or null.
         * @return exit status string
         */
        public String getExitStatus();
        /**
         * The setExitStatus method assigns the user-specified exit status for
         * the current flow. When the flow ends, the exit status of the flow is
         * the value specified through setExitStatus. If setExitStatus was not
         * called or was called with a null value, then the exit status
         * defaults to the batch status of the flow.
         * @Param status string
         */
```

```
	public void setExitStatus(String status);
}
```

6.8.4 SplitContext

```
package javax.batch.runtime.context;
/**
  *
  * SplitContext is a class field type associated with the @BatchContext
  * annotation. A SplitContext provides information about the current
  * Split execution.
  *
  * @see javax.batch.annotation.context.BatchContext
  */

public interface SplitContext {
	/**
	 * The getId method returns the current split's identity.
	 * @return split id string
	 */
	public String getId();
	/**
	 * The getBatchStatus method returns the current batch status of the
	 * current job. This value is set by the batch runtime and changes as
	 * the batch status changes.
	 * @return batch status string
	 */
	public String getBatchStatus();
	/**
	 * The getExitStatus method simply returns the exit status value stored
	 * into the Split context through the setExitStatus method or null.
	 * @return exit status string
	 */
	public String getExitStatus();
	/**
	 * The setExitStatus method assigns the user-specified exit status for
	 * the current Split. When the Split ends, the exit status of the Split
	 * is
	 * the value specified through setExitStatus. If setExitStatus was not
	 * called or was called with a null value, then the exit status
	 * defaults to the batch status of the Split.
	 * @Param status string
	 */
	public void setExitStatus(String status);
}
```

6.8.5 PartitionPlan

```
package javax.batch.parallel;
/**
 *
```

```
 * PartitionPlan is a helper class that carries partition processing
 * information set by the @PartitionAlgorithm method.
 *
 * A PartitionPlan contains:
 * <ol>
 * <li>number of subjobs to dispatch for a partitioned step of flow</li>
 * <li>substitution properties for each subjob comprising the partitioned
 * step or flow</li>
 * </ol>
 *
 * @see javax.batch.annotation.parallel.PartitionAlgorithm
 */
import java.io.Externalizable;
import java.util.Properties;

public interface PartitionPlan extends Externalizable {

/**
 * Set count of subjobs.
 * @param count specifies the subjob count
 */
public void setSubJobCount(int count) ;

/**
 * Sets array of substitution Properties objects for the set of subjobs.
 * @param props specifies the Properties object array
 */
public void setSubJobProperties(Properties[] props) ;

/**
 * Gets count of subjobs.
 * @return subjob count
 */
public int getSubJobCount() ;

/**
 * Gets array of subjob Properties objects for subjobs.
 * @return subjob Properties object array
 */
public Properties[] getSubJobProperties() ;

}
```

6.8.6 JobOperator

```
package javax.batch.operations;

import java.util.List;
import java.util.Set;
import java.util.Map;
import java.util.Properties;

public interface JobOperator {
```

```java
/**
 * Returns all executionIds belonging to a particular job instance.
 * @param instanceId identifies the job instance
 * @return List of executionIds
 * @throws NoSuchJobInstanceException
 */
List<Long> getExecutions(long instanceId) throws
      NoSuchJobInstanceException;

/**
 * Returns number of instances of a job with a particular name.
 * @param jobName specifies the name of the job.
 * @return count of instances of the named job.
 * @throws NoSuchJobException
 */
int getJobInstanceCount(String jobName) throws NoSuchJobException;

/**
 * Returns all instanceIds belonging to a job with a particular name.
 * @param jobName identifies the job name.
 * @param start identifies the relative starting number to return from
 * the maximal list of job instances.
 * @param count identifies the number of instance ids to return from
 * the starting position of the maximal list of job instances.
 * @return list of instance ids
 * @throws NoSuchJobException
 */
List<Long> getJobInstances(String jobName, int start, int count) throws
      NoSuchJobException;

/**
 * Returns executionIds for all running executions across all instances
 * of a job with a particular name.
 * @param jobName identifies the job name.
 * @return a Set of executionIds
 * @throws NoSuchJobException
 */
Set<Long> getRunningExecutions(String jobName) throws
      NoSuchJobException;

/**
 * Returns job parameters for specified execution.  These are the
 * key/value pairs specified when the instance was started or
 * restarted.
 * @param executionId identifies the execution.
 * @return a Properties object containing the key/value job parameter
 * pairs.
 * @throws NoSuchJobExecutionException
 */
Properties getParameters(long executionId) throws
      NoSuchJobExecutionException;

/**
 * Creates a new job instance and starts the first execution of
 * that instance.
 * @param job specifies the Job XML describing the job.
 * @param jobParameters specifies the keyword/value pairs for
```

```java
 * property override and substitution in Job XML.
 * @return initial executionId of the new job instance.
 * @throws JobStartException
 */
Long start(String job, Properties jobParameters) throws
      JobStartException;

/**
 * Restarts a failed or stopped job instance.
 * @param executionId belonging to the instance to restart.
 * @param jobParameters specify replacement job parameters for the
 * job restart.
 * The replacement add to and/or override the original job
 * parameters that were specified when the instance was
 * originally started.
 * @return new executionId
 * @throws JobInstanceAlreadyCompleteException
 * @throws NoSuchJobExecutionException
 * @throws NoSuchJobException
 * @throws JobRestartException
 */
Long restart(long executionId, Properties jobParameters) throws
      JobInstanceAlreadyCompleteException,
      NoSuchJobExecutionException, NoSuchJobException,
      JobRestartException;

/**
 * Request a running execution stops.
 * @param executionId specifies the execution to stop.
 * @return true if the execution stopped;  false if it did not.
 * @throws NoSuchJobExecutionException
 * @throws JobExecutionNotRunningException
 */
boolean stop(long executionId)throws NoSuchJobExecutionException,
      JobExecutionNotRunningException;

/**
 * Return an execution summary for the specified execution.
 * @param executionId specifies the execution.
 * @return an execution summary string.
 * @throws NoSuchJobExecutionException
 */
String getSummary(long executionId) throws NoSuchJobExecutionException;

/**
 * Returns step summaries for all steps of a specified execution.
 * @param executionId specifies the execution.
 * @return a map of all step summaries for the specified execution.
 * @throws NoSuchJobExecutionException
 */
Map<Long, String> getStepExecutionSummaries(long executionId) throws
      NoSuchJobExecutionException;

/**
 * Returns a set of all job names known to the batch runtime.
 * @return a set of job names.
 */
```

```
  Set<String> getJobNames();


  /**
   * Return the job instance for the specified job instance id
   * @param instanceId specifies the requested job instance
   * @return job instance
   */
  JobInstance getJobInstance(long instanceId);

  /**
   * Return all job executions belonging to the specified job instance
   * @param jobInstance specifies the job instance
   * @return list of job executions
   */
  List<JobExecution> getJobExecutions(long instanceId);

  /**
   * Return job execution for specified execution id
   * @param executionId specifies the requested job execution
   * @return job execution
   */
  JobExecution getJobExecution(long executionId);

  /**
   * Return step execution for specified execution
   * @param jobExecutionId specifies the job execution
   * @param stepExecutionId specifies the step belonging to that
   * execution
   * @return step execution
   */
StepExecution getStepExecution(long jobExecutionId, long
  stepExecutionId);

/**
 * Return set of job instance that have specified job name
 * @param jobName specifies name of the job
 * @return set of job instances
 */
Set<JobInstance> findRunningJobInstances(String jobName);


  /**
   * Registers a job end callback. A job end callback is invoked by
   * the batch runtime when a job ends.
   * @param callback specifies the callback instance.
   * @return a callBack id.
   * @throws CallbackRegistrationException
   */
  long registerJobEndCallback(JobEndCallback callback) throws
        CallbackRegistrationException;

  /**
   * Deregisters a job end callback.
   * @param callbackId specifies the callback to deregister.
   * @throws CallbackDeregistrationException
   */
```

```
    void deregisterJobEndCallback(long callbackId) throws
          CallbackDeregistrationException;

}
```

7 Credits

Special thanks to the members of the JSR 352 Expert Group for contributing the concepts and ideas found in this specification draft. Membership: http://jcp.org/en/jsr/detail?id=352

Section 3 Domain Language of Batch, adapted from Spring Batch Reference Documentation:

http://static.springsource.org/spring-batch/trunk/reference/html-single/index.html