**WebSphere Application Server**

# WebSphere Liberty Batch Migration Guide

*Version Date:* May 16, 2016

See "Document change history" on page 23 for a description of the changes in this version of the document

IBM Software Group
Application and Integration Middleware Software

Written by:
**Mary Curran**
IBM RTP
`mcurran@us.ibm.com`

## Table of Contents

## Introduction

WebSphere Compute Grid has been around since WebSphere Application Server V6.0, first as a stack product and, as of WAS V8.5, as a part of the application server itself. WebSphere Compute Grid provides a programming framework and an execution runtime environment for batch programming which alleviates much of the programming effort required and allows developers to focus on the business logic of their applications. However, it was built on a proprietary model and not on an open standard.

Starting with WebSphere Liberty V8.5.5.6, support for the Java EE7 standard is supported, including JSR-352, the open standard for Java Batch. The support in WebSphere Liberty also includes operational enhancements which are outside of the scope of JSR-352 and align with some of the operational aspects of WebSphere Compute Grid.

WebSphere Liberty Batch and WebSphere Compute Grid do share commonalities. Jobs in both consist of steps which reference Java artifacts packaged in Java Enterprise applications. Jobs are defined using XML. Both provide job logging, parallel job processing and a mechanism for remote execution. However, since WebSphere Liberty Batch is built on an open standard, there are differences that have to be addressed when moving from WebSphere Compute Grid to WebSphere Liberty Batch. There are also additional capabilities that can be used. This paper is designed to address both areas and ease the transition to WebSphere Liberty Batch.

## Job XML

Jobs are expressed in WebSphere Liberty Batch and WebSphere Compute Grid using XML. The Job XML Language in WebSphere Liberty Batch is referred to as JSL or Job Specification Language.

The following table illustrates both the similarities and differences of the Job XML language for WebSphere Compute Grid and WebSphere Liberty Batch.

| **Compute Grid xJCL element** | **Liberty Batch JSL element** | **Liberty Batch Comments** |
|---|---|---|
| job | job | Identifies a job. Same in both environments. |
| jndi-name | n/a | No job controller stateless session bean needed. |
| props<br>prop | properties<br>property | Child element of job, step, decision, reader, processor, writer, batchlet, listener, etc. |
| substitution-props<br>prop | The value portion of any attribute:<br><br>"#{jobParameters['***target***']} ?:***default;***" | Supports substitution as part of any attribute value. A substitution expression may reference a job parameter by specifying the name of the parameter in the jobParameters substitution expression operator. |
| checkpoint-algorithm<br>classname | checkpoint-algorithm ref="***name***" | Specifies an optional custom checkpoint-algorithm. It is a child element of the chunk element. |
| Results-algorithms<br>result-algorithm | n/a | Exit status replaces job and step return codes. |
| listener | listeners<br>listener ref="***name***" | Specified at the job or step level. ***name*** specifies the name of the batch artifact that implements the listener.<br><br>Step level listeners:<br>chunk step – step listener, item read listener, item process listener, item write listener, chunk listener, skip listener, and retry listener<br><br>batchlet step – step listener |

| Compute Grid xJCL element | Liberty Batch JSL element | Liberty Batch Comments |
|---|---|---|
| job-step<br>classname | step id=”***id***” next=”***target***” | No classname specified. See section Batch Processing Loop.<br><br>***id*** specifies the logical name of the step.<br>***target*** is an optional attribute that specifies the next step, flow, split, or decision to run after this step completes.<br><br>If the ***next*** attribute is not specified and there are no matching transition elements, the job will terminate after this step. |
| batch-data-streams<br>bds<br>logical-name<br>impl-class | chunk<br>reader ref=”***name***”<br>processor ref=”***name***”<br>writer ref=”***name***” | reader specifies the item reader for a chunk step; limited to one per step.<br><br>processor specifies the item processor for a chunk step; limited to one per step.<br><br>writer specifies the item writer for a chunk step; limited to one per step.<br><br>***name*** specifies the batch artifact that implements the reader, processor or writer.<br><br>Data aggregation for reads or writes across different data streams is outside of the scope of JSR 352 and is the responsibility of the application. |
| run instances=”multiple” | partition | Specifies that a step is a partitioned step. Child element of a step element. |
| step-scheduling | decision id=”***id***” ref=”***ref-name***” | Provides customized way of determining sequencing among steps, flows, and splits. See section Execution Directives.<br><br>***id*** specifies the logical name of the decision and is used for identification purposes.<br><br>***ref*** specifies the name of the batch artifact that implements the decision. |

| Compute Grid xJCL element | Liberty Batch JSL element | Liberty Batch Comments |
|---|---|---|
| n/a | flow id=”***id***” next=”***flow-id \| step-id \| split-id \| decision-id***” | A flow defines a sequence of execution elements that execute together as a unit. See section Parallel Processing with Splits and Flows (New).<br><br>***id*** specifies the logical name of the flow and is used for identification purposes.<br><br>***xxxx-id*** specifies the next step, flow, split or decision to run after this step is complete. |
| n/a | split id=”***name***” next=”***flow-id \| step-id \| split-id \| decision-id***” | A split defines a set of flows that execute concurrently. See section Parallel Processing with Splits and Flows (New).<br><br>***id*** specifies the logical name of the split and is used for identification purposes.<br><br>***xxxx-id*** specifies the next step, flow, split or decision to run after this step is complete. |

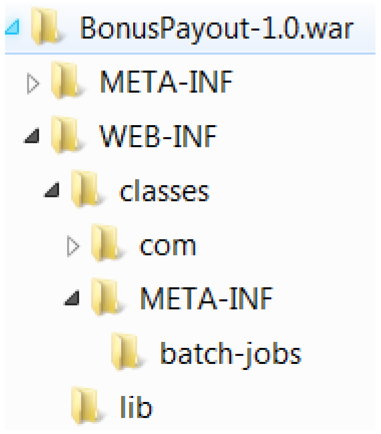
## Packaging

Unlike WebSphere Compute Grid applications whose batch artifacts are packaged into Enterprise Archive (EAR) files and require a Batch Controller Bean, WebSphere Liberty Batch applications require no special packaging. The batch artifacts can be included inside any Java archive type supported by WebSphere Liberty (e.g. war, ear). The application is deployed by either dropping the application into the WebSphere Liberty server's dropins directory or by adding an application entry to the server configuration.

If the JSL document is packaged as part of the application, it is stored under the META-INF/batch-jobs directory.

- EJB jar files - META-INF/batch-jobs
- war files – WEB-INF/classes/META-INF/batch-jobs

Example:



```
BonusPayout-1.0.war
  META-INF
  WEB-INF
    classes
      com
      META-INF
        batch-jobs
    lib
```

## Batch Artifact References

Batch artifact references can be resolved in one of the following ways.

### CDI Bean Name

The CDI loader can be used to load batch artifacts by specifying the bean name (@Named value) in the reference (*ref*) attribute of the batch artifact in the JSL.

### Entry in batch.xml

A batch application can use the archive loader to load batch artifacts.  This is accomplished by supplying an optional batch.xml file.  The batch.xml file is stored under the META-INF directory (WEB-INF/classes/META-INF for .war files) and contains the reference name and the fully qualified class name of the batch artifact. The reference in the JSL is then resolved by looking up the reference name in the batch.xml.

The format and content of the batch.xml file is as follows:

```
<batch-artifacts xmlns=”http://xmlns.jcp.org/xml/ns/javaee” >
    <ref id=”<reference-name>” class=”<impl-class-name>” />
</batch-artifacts>
```

### Fully-qualified Class Name

The thread context class loader is used to load a batch artifact if the fully qualified class name is specified in the reference (*ref)* attribute of the batch artifact.

## Status

A WebSphere Liberty Batch job contains three separate “status” values: Batch Status, Instance State and Exit Status.  The Job Status in WebSphere Compute Grid is a combination of both Batch Status and Instance State, containing values found in both. The equivalent of Exit Status in WebSphere Compute Grid is the user-defined return code value.  The difference being that the former is a string value and the latter, an integer value.

### Batch Status

Batch status is a job and step runtime status value set by the batch runtime.
The Batch Status values are:  STARTING, STARTED, STOPPING, STOPPED, FAILED, COMPLETED, ABANDONED

### Instance State

Instance State contains the current runtime state of the Job Instance.
The Instance State values are:  SUBMITTED, JMS_QUEUED, JMS_CONSUMED, DISPATCHED, FAILED, STOPPED, COMPLETED, ABANDONED

### Exit Status

Exit Status is a user-defined value for jobs, steps and partitions which is set through either the Job XML or by the batch application. It is the value that is used in the Job XML transition elements.

## Execution Directives

In WebSphere Compute Grid only sequential step execution is supported. However, the ability does exist to create return code-based conditional flows to determine whether or not a step is invoked while processing a batch job. This ability to conditionally influence sequencing among elements has been expanded in WebSphere Liberty Batch through the Transition and Decision elements.

### Transition element

Step execution in WebSphere Liberty Batch is not required to be sequential. The transition and decision elements provide a more flexible approach to step execution. The transition element is specified in the JSL in the containment scope of a step, flow or decision. It can be used to direct the job execution sequence or terminate job execution based on the exit status of the containing element. There are four types of transition elements as listed below.

- next - directs the execution flow to the next specified execution element.
- fail - causes a job to end with a batch status of FAILED.
- end - causes a job to end with a batch status of COMPLETED.
- stop – causes a job to end with a batch status of STOPPED. Can also be used to specify where to restart the job.

Example:

```
<step id=”Step1” next=”Step2”>
    <next on “RC4” to “Step3”/>
    <fail on “RC_ABORT” exist-status=”ABORTED” />
</step>
```

In the above example, Step1 will transition to Step2, unless an exit status of RC4 or RC_ABORT is received, in which case the job will either go to Step3 or fail with an exit status of ABORTED.

Transition elements are always evaluated first and in sequential order as they occur within the JSL document. The first one that matches performs the corresponding transition and the rest of the transition elements are ignored.

### Decision element

The decision element provides a customized way of determining sequencing among steps, flows and splits and can be used to determine the batch exit status. It can be specified as the target of the "next" attribute from a step, flow, split or another decision. The decision element must supply a batch artifact which implements the Decider interface. The String value returned becomes the exit status value on which the next transition is chosen. The decision element uses any of the transition elements to select the next transition.

Example:

```
<decision id=decider1" ref="transitionDecider">
   <stop on="JOB_STOP_STATUS" />
   <fail on="JOB_FAIL_STATUS" />
   <next on="STEP_CONTINUE" to "flow1" />
</decision>
```

## Operational Characteristics

### Batch Processing Loop

The batch processing loop is the method by which processing occurs within a step.

### Chunk step

The chunk step is the equivalent of the batch step in WebSphere Compute Grid although the batch processing loop is somewhat different. In both cases the batch processing loop is an iterative loop through the data, with periodic checkpoints taken of the input/output streams and periodic commits of data written out during processing. The following bullets illustrate some of the key differences.

- The chunk step processing loop is the same as what is contained in WebSphere Compute Grid's GenericXDBatchStep (read, process, write), but is included in the container instead of in a separate framework.
- The container code invokes ItemReader readItem, ItemProcessor processItem and eventually ItemWriter writeItems in the loop.
- The only code required to be written is the implementations of the ItemReader, ItemProcessor and ItemWriter interfaces. There is no longer any code required to be written for the Step.

- Reader and writer methods to be implemented are open, close, readItem, writeItem and checkpointInfo.
- The previous checkpoint information, if any, is passed in on open.
- Only one reader and one writer is supported.

### Batchlet Step

The Batchlet Step's processing is the same as the processing for a Compute Intensive (CI) step in WebSphere Compute Grid. The step is called, runs and ends. If a stop is issued, the Batchlet's stop() method is invoked by the batch runtime. This is equivalent to the release() method in WebSphere Compute Grid.

If using a "stop" flag to signal the running process to stop, the "stop" flag should be marked volatile.

### Checkpoints

The default JSR-352 specification defined checkpoint algorithm is the same as WebSphere Compute Grid's TimeRecordBased algorithm. Like WebSphere Compute Grid, an API is provided for building additional custom checkpoint algorithms. Custom checkpoint algorithms are specified in the JSL.

### Restart Processing

A WebSphere Liberty Batch job is eligible to be restarted if the job completes with a batch status of STOPPED or FAILED. In WebSphere Compute Grid, a job could not be restarted with a status of EXECUTION_FAILED. Only jobs in CANCELLED or RESTARTABLE status could be restarted.

In WebSphere Liberty Batch, a new execution is created for each restart. An entirely new set of job parameters can be specified or a delta set of job parameters can be specified by using the *reusePreviousParams* option.

The execution sequence on restart proceeds according to a loop defined by a few simple rules:

1. **Start by transitioning to the initial step.**
   By default, this is the same initial step as upon the initial "start" job execution (the first child on reading the job XML document sequentially from top-to-bottom). However, this will be overridden when the job is stopped via a <*stop*> transition

element with a *restart* attribute value supplied. In this special case, the initial step upon restart will be the value of this *restart* attribute

2. **Decide whether or not to execute the current step**.
   By default, execute or possibly re-execute the current step unless it has already been completed. You can override this default behavior and execute a step unconditionally by declaring the step with the *allow-restart-if-complete* attribute set to “true”.

3. **Transition based on exit status:**
   If the step on the current restart execution is not executed/re-executed, transition based on the previous exist status. If the step is executed/re-executed, transition based on the new step execution’s exit status.

   Once the transition occurs, the flow either goes back to step 2, where the decision is made as to whether or not to execute the new “current step”, or eventually the job is terminated.

The rules above should make it clear that in this context “transitioning to a step” involves the batch container moving a “cursor” to the next candidate step and then deciding whether or not to execute the step according to the rules just described.

The other execution elements are handled during a restart in a straightforward manner:

- If the current execution element is a decision, execute the decision unconditionally.
- If the current execution element is a flow, transition to the first execution element in the flow and repeat the process to determine if the current execution element should be re-executed. For splits, the flow processing is followed in parallel for each flow in the split.

Please see the JSR 352: Batch Applications for the Java Platform specification for additional details.

## Skip/Retry Processing

The skip/retry support in WebSphere Liberty Batch is built into the container and does not require a special framework as in WebSphere Compute Grid.

**WebSphere Liberty Batch**:
- The total number of retry attempts is set by the *retry-limit* attribute on the chunk element. The default *retry-limit* value is unlimited retries.
- The total number of skip attempts is set by the *skip-limit* attribute on the chunk element. The default *skip-limit* value is unlimited skips.

- Skippable and retryable exception classes are specified in the JSL as a child element of the chunk element

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


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

- Include and exclude elements are **not** mutually exclusive. In the following example, all exceptions are skipped except for java.io.FileNotFoundException and any subclasses of it.
  Example:

```
<skippable-exception-classes>
    <include class="java.lang.Exception"/>
    <exclude class="java.io.FileNotFoundException"/>
</skippable-exception-classes>
```

- There is no retry delay time that can be specified as in WebSphere Compute Grid.
- Retry/skip processing applies to exceptions thrown from the reader, processor, or writer batch artifacts of a chunk type step.
- Default retry behavior is to rollback the current chunk and reprocess it with an item count of 1 and a checkpoint policy of item.
- The default retry behavior can be overridden with the *no-rollback-exception-classes* element. Any entries in the *no-rollback-exception-classes* element must also exist in the *retryable-exception-classes element*. If rollback does not occur, the identical operation that caused the exception on the read, process or write is retried.

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

**WebSphere Compute Grid**:

- The total number of retry attempts is set by the *com.ibm.batch.step.retry.count* property at the job-step level.

- The total number of skip attempts is set by the *com.ibm.batch.bds.skip.count* property at the batch data stream level.
- Skippable and retryable exception classes are specified in the xJCL as a property of the batch data stream (*bds*) element.

```
<props>
   <prop name=”com.ibm.batch.bds.skip.count” value=”{value}”/>
   <prop name=”com.ibm.batch.bds.skip.include.exception.class.1”
            value=”{class name}” />
   <prop name=”com.ibm.batch.bds.skip.include.exception.class.2”
            value=”{class name}” />
</props>

<props>
   <prop name=”com.ibm.batch.step.retry.count” value=”{value}”/>
   <prop name=”com.ibm.batch.step.retry.delay.time” value=”{value}” />
   <prop name=”com.ibm.batch.step.retry.exclude.exception.class.1”
            value=”{class name}” />
</props>
```

- If the “include exception” property is not specified, the default is to include all exceptions.
- The retry behavior is to rollback uncommitted transactions to the last checkpoint interval and retry the step.

## Context

In WebSphere Compute Grid, there is only one context object, JobStepContext, which is a combination of both Job and Step Contexts. In WebSphere Liberty Batch, the Job and Step contexts were separated into 2 context objects: JobContext and StepContext.

### Job Context

The Job Context provides information about the current job execution. There is a distinct Job Context for each sub-thread of a parallel execution (e.g. partitioned step). The following list provides the available getters/setters along with their corresponding WebSphere Compute Grid equivalent.

| WebSphere Liberty Batch | WebSphere Compute Grid |
|---|---|
| getJobName | getJobID [1] |
| getTransientUserData | getJobLevelTransientUserData |
| setTransientUserData | setJobLevelTransientUserData |

| | |
|---|---|
| getInstanceId | getJobID [1] |
| getExecutionId | n/a |
| getProperties | getJobLevelProperties |
| getBatchStatus | n/a |
| getExitStatus | getReturnCode |
| setExitStatus | setReturnCode |

[1] JobID consists of both Job Name and ID.

### Step Context

The Step Context provides information about the current step of a job execution. For a partitioned step, there is a distinct Step Context for the parent thread and for each sub-thread.

| WebSphere Liberty Batch | WebSphere Compute Grid |
|---|---|
| getStepName | getStepID |
| getTransientUserData | getStepLevelTransientUserData |
| setTransientUserData | setStepLevelTransientUserData |
| getStepExecutionId | n/a |
| getProperties | n/a |
| getPersistentUserData | getJobLevelPersistentUserData |
| setPersistentUserData | setJobLevelPersistentUserData |
| getBatchStatus | n/a |
| getExitStatus | getReturnCode |
| setExitStatus | setReturnCode |
| getException | getUserException |
| getMetrics | getRecordMetrics |

## Listeners

In WebSphere Compute Grid, there are three listeners that can be implemented which are invoked at various stages in the lifecycle of the job. These listeners and their corresponding interfaces are the job level listener (JobListener), the retry listener (RetryListener) and the skip listener (SkipListener). As well as those mentioned above, WebSphere Liberty Batch supports additional listeners providing more access to lifecycle events. The following describes the listeners in detail.

## Job Listener

A job listener is specified at the job level in the JSL and is used to intercept job execution events. It implements the JobListener interface and receives control before job execution begins (beforeJob) and after job execution ends (afterJob). It is a subset of the JobListener interface in WebSphere Compute Grid which also contains step level events which are now contained in the Step Listener.

## Step Listener

A step listener is specified at the step level in the JSL and is used to intercept step execution events. It implements the StepListener interface and receives control before step execution begins (beforeStep) and after step execution ends (afterStep).

## Item Read Listener

An item read listener is specified at the step level in the JSL and is used to intercept item reader processing events. It implements the ItemReadListener interface and receives control before an item is read (beforeRead), after an item is read (afterRead) and after an item reader throws an exception in the readItem method (onReadError).

## Item Process Listener

An item process listener is specified at the step level in the JSL and is used to intercept item processing events. It implements the ItemProcessListener interface and receives control before an item is processed (beforeProcess), after an item has been processed (afterProcess) and after an item processor throws an exception in the processItem method (onProcessError).

## Item Write Listener

An item write listener is specified at the step level in the JSL and is used to intercept item writer processing events. It implements the ItemWriteListener interface and receives control before an item is written (beforeWrite), after an item is written (afterWrite) and after an item writer throws an exception in the writeItems method (onWriteError).

### Chunk Listener

A chunk listener is specified at the step level in the JSL and is used to intercept chunk processing events. It implements the ChunkListener interface and receives control before processing begins on a chunk (beforeChunk), after processing ends on a chunk (afterChunk) and before the chunk transaction is rolled back (onError).

### Skip Listener

A skip listener is specified at the step level in the JSL and is used to intercept skippable exceptions from an item reader, an item processor or an item writer. Unlike WebSphere Compute Grid which has one SkipListener interface and only supports skips on reads and writes, WebSphere Liberty Batch has three skip listener interfaces: SkipReadListener, SkipWriteListener and SkipProcessListener. A skip listener receives control when a skippable exception is thrown from an item reader's readItem method (onSkipReadItem), when a skippable exception is thrown from an item writer's writeItems method (onSkipWriteItem) and when a skippable exception is thrown from an item processor's processItem method (onSkipProcessItem).

### Retry Listener

A retry listener is specified at the step level in the JSL and is used to intercept retry processing events from an item reader, an item processor or an item writer. Unlike WebSphere Compute Grid which has one RetryListener interface, WebSphere Liberty Batch has three retry listener interfaces: RetryReadListener, RetryWriteListener and RetryProcessListener. A retry listener receives control when a retryable exception is thrown from an item reader's readItem method (onRetryReadException), when a retryable exception is thrown from an item writer's writeItems method (onRetryWriteException) and when a retryable exception is thrown from an item processor's processItem method (onRetryProcessException).

## Parallel Processing with Partitions

Parallel processing in WebSphere Liberty Batch is accomplished through the use of partitioned steps. Partitioned steps look and function very much like parallel steps in WebSphere Compute Grid. One difference to note is that partitioned steps do not appear as separate sub-jobs and cannot have operations performed directly against them.

## PartitionMapper

The number of partitions and the number of threads on which to execute the partitions of the step is controlled through either a static specification in the JSL (*plan* element) or through a batch artifact called a partition mapper.

```
<partition>
   <plan partitions=”{number}” threads=”{number}”>
            <properties partition=”partition-number”>
                     <property name=”{property-name}” value=”{value}” />
            </properties>
   </plan>
</partition>

                              OR
<partition>
   <mapper ref=”MyStepPartitioner”/>
</partition>
```

The *partitions* attribute of the plan element is analogous to the *parallel.jobcount* property specified as a child of the run element in WebSphere Compute Grid, just as the mapper is closely aligned with the Parallel Job Manager Parameterizer API in WebSphere Compute Grid.

When defining a statically partitioned step, you can specify unique property values to pass to each partition in the JSL using the *property* element as shown.

## PartitionReducer

The PartitionReducer provides programmatic control over the logical transaction of the partitioned step. The logical transaction provides a unit-of-work scope across all partitions of a partitioned step. The PartitionReducer is used to process logical transaction lifecycle events such as beginPartitionedStep, beforePartitionedStepCompletion, rollbackPartitionedStep and afterPartitionedStepCompletion.

The WebSphere Compute Grid equivalent of the PartitionReducer is the Parallel Job Manager Synchronization API.

```
<partition>
   <reducer ref=”MyStepPartitionReducer”/>
</partition>
```

## PartitionCollector

The PartitionCollector is used to send intermediary results for analysis from each partition to the step's PartitionAnalyzer. A separate PartitionCollector instance runs on each thread executing a partition of the step. It is invoked after each checkpoint and at the end of the partition.

The WebSphere Compute Grid equivalent of the PartitionCollector is the Parallel Job Manager SubJobCollector API.

```
<partition>
   <collector ref=”MyStepCollector”/>
</partition>
```

## PartitionAnalyzer

The PartitionAnalyzer receives intermediary results sent from each partition via the PartitionCollector. It runs on the step's main thread and serves as a collection point for data sent from the partitions. It also receives control at the end of each partition with the partition's batch and exit statuses. The PartitionAnalyzer provides the ability to determine the overall outcome of the step based on data received from each partition.

The WebSphere Compute Grid equivalent of the PartitionAnalyzer is the Parallel Job Manager SubJobAnalyzer API.

```
<partition>
   <analyzer ref=”MyStepAnalyzer”/>
</partition>
```

## Restart

On restart, the PartitionMapper determines whether or not the partitions used in the previous execution of the step will or will not be used within the current execution of the step. If they are not used, all partition execution data including checkpoints, persistent user data, etc. from the earlier execution are discarded and new partitions are executed. The default behavior is to execute or re-execute all incomplete partitions using previous partition execution data. This behavior can be overridden by the PartitionMapper using the PartitionPlan's override method, setPartitionsOverride. If set to true, all results from previous partitions are discarded. This is a more limited option of restart behavior than what existed in WebSphere Compute Grid, but encapsulates the most common behavior used.

## Parallel Processing with Splits and Flows (New)

Split/Flow processing allows parallel processing with different actions.  There is no equivalent function in WebSphere Compute Grid.

### Flow

A flow defines a sequence of execution elements that execute together as a unit.  When the flow is finished, the entire flow transitions to the execution element specified by the *next* attribute which may be a step, split, decision or another flow.  A flow may contain step, flow, decision and split execution elements.

```
<flow id="{name}" next="{flow-id | step-id | split-id | decision-id}">
    <step>…</step>
    …
    <step>…</step>
</flow>
```

### Split

A split defines a set of flows that execute concurrently.  A split may only include flow elements as children and each flow runs on a separate thread.  When the split is finished, the entire split transitions to the next execution element which may be a step, flow, decision, or another split.

```
<split id="{name}" next="{flow-id | step-id | split-id | decision-id}">
    <flow>…</flow>
    …
    <flow>…</flow>
</split>
```

## Job Repository

The Job Repository holds information about jobs such as instance and execution IDs, batch status, start time and last update time, much the same as the information stored in the external job database in WebSphere Compute Grid.  However, in addition to the file-based Derby and relational database support provided by both, WebSphere Liberty Batch provides an in-memory job repository for development and test environments that do not require data to be persisted across server starts.

## Job Logs

In both WebSphere Compute Grid and WebSphere Liberty Batch, a log is written for each job which contains output from both the runtime and from the application. In WebSphere Liberty Batch, job logs are only created on the Executors (i.e. Endpoints). There are no job logs on the Dispatchers (i.e. Schedulers). Furthermore, applications write messages to the job logs using java.util.logging in contrast to WebSphere Compute Grid's use of System.out and System.err.

## JobOperator (New)

The JobOperator interface provides a means of starting, stopping, restarting and retrieving information about jobs. It is accessed using the factory method, getJobOperator, in the BatchRuntime class and can be useful in a development or testing environment.
Although query operations can be performed across the domain as defined by the persistent store, start/stop/restart operations are local to the application in which they are contained.

## Document change history

Check the version date on the title page for the version of the document.

| | |
|---|---|
| *March 25, 2016* | Initial Version |
| *April 25, 2016* | Review comments incorporated |
| *May 3, 2016* | Minor revisions and addition of job log section. |
| *May 16, 2016* | Final Version |