**Spring Batch**

*“*Batch processing is the execution of a series of programs (“jobs”) on a computer without manual intervention”

**Spring Batch** is an open source framework for batch processing – execution of a series of jobs. Spring Batch provides classes and APIs to read/write resources, transaction management, job processing statistics, job restart and partitioning techniques to process high-volume of data.

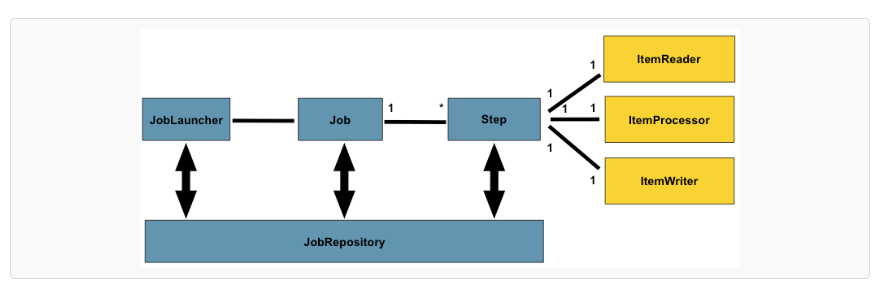
**Usage Scenarios:**

* **Conversion Applications:** Applications that convert input records into the required structure or format.
* **Filtering or validation applications**
* **Database extractors:** Applications that read data from a database or input files and write the desired filtered data to an output file or to other database. There are also applications that updates large amounts of data in the same database where the input records come from.
* **Reporting:** Applications that read large amounts of data from a database or input files, process this data and produce formatted documents based on that data that are suitable for printing or sending via other systems.

**Technical Features:**

* Batch developers use the Spring programming model: concentrate on business logic; let the framework take care of infrastructure.
* Clear separation of concerns between the infrastructure, the batch execution environment, and the batch application.
* Provides common, core execution services as interfaces that all projects can implement.
* Provide simple and default implementations of the core execution interfaces that can be used ‘out of the box’.
* Easy to configure, customize, and extend services, by leveraging the spring framework in all layers.
* All existing core services should be easy to replace or extend, without any impact to the infrastructure layer.
* Provides a simple deployment model, with the architecture JARs completely separate from the application, built using Maven.

**Architecture:**

****

**Job:** A job is an entity that encapsulates an entire batch process. As is common with other Spring projects, a  job will be wired together via an XML configuration file or Java based configuration.

**Step**: A Step is a domain object that encapsulates an independent, sequential phase of a batch job and contains all of the information necessary to define and control the actual batch processing.

Steps can be processed in either of the following two ways.

Chunk and Tasklet

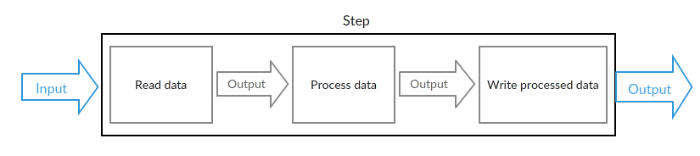
**Tasklet**:  the job to be run a single granular task like executing a query or deleting a resource or invoking procedure calls then tasklet processing is used.

**Chunk**: job to be run is complex and involves executing of tasks involving reads, processing and writes the we use chunk oriented processing. Usually used in scenarios where multiple aggregated steps need to be run like copying, processing and transferring of data. It’s the most common way of executing a step.

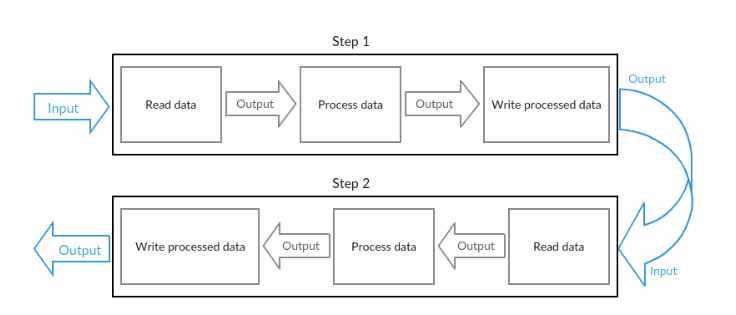
A Spring Batch job consists of the following components:

* The Job represents the Spring Batch job. Each job can have one or more steps.
* The Step represents an independent logical task (i.e. import information from an input file). Each step belongs to one job.
* The ItemReader reads the input data and provides the found items one by one. An ItemReader belongs to one step and each step must have only one ItemReader.
* The ItemProcessor transforms items into a form that is understood by the ItemWriter one item at a time. An ItemProcessor belongs to one step and each step can have one ItemProcessor.
* The ItemWriter writes an information of an item to the output one item at a time. An ItemWriter belongs to one step and a step must have only one ItemWriter.
* In commit-interval the number of items that are processed within a commit can be configured.
* Transaction-manager - Spring's PlatformTransactionManager that will be used to begin and commit transactions during processing.

**Simple Job with one step:**



**Simple job with two steps:**



Sample Job Configuration:

<job id="jobid1">

<step id="stepid1” next="stepid2"/>

<step id="stepid2">

<!— define the chunks ,tasklets here -->

</step>

</job>

**Job Repository:**  The JobRepository that will be used to periodically store the StepExecution and ExecutionContext during processing (just before committing). For an in-line <step/> (one defined within a <job/>) it is an attribute on the <job/> element; for a standalone step, it is defined as an attribute of the <tasklet/>.

**Job Launcher:** Job Launcher is an interface and .Implementations of its run() method take care of starting job executions for the given jobs and job parameters.

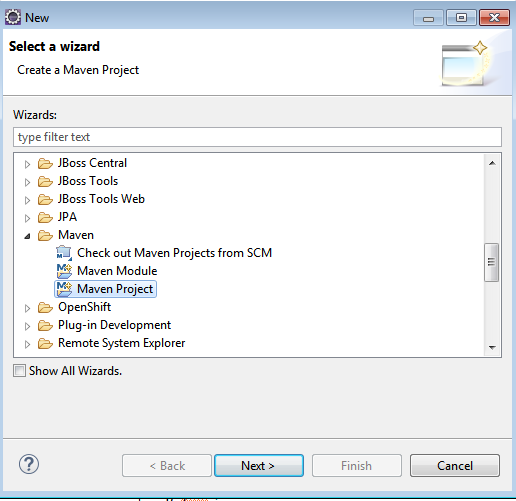
**Sample Program to copy the data from csv file to xml :**

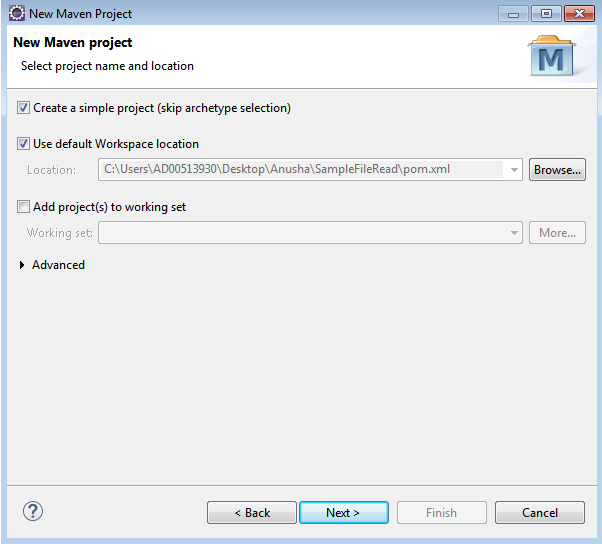
**Prerequisites**

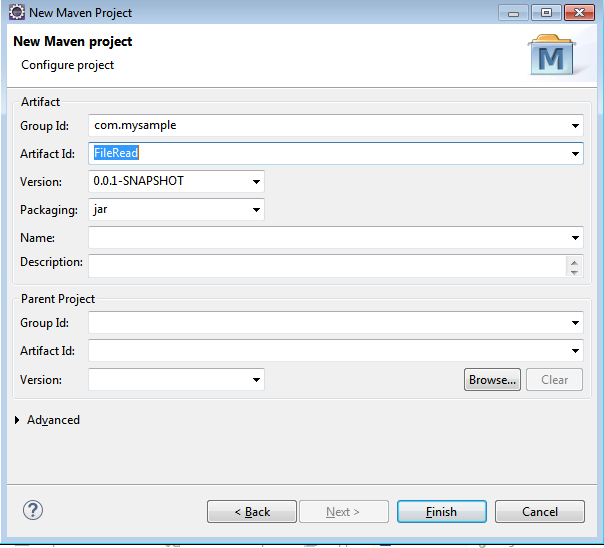
We must have installed the following

1. Maven 3 or later
2. Eclipse 4.2
3. JDK 1.6 or higher version
4. Spring Core 3.2.2.RELEASE
5. Spring OXM 3.2.2.RELEASE
6. Spring JDBC 3.2.2.RELEASE
7. Spring Batch 2.2.0.RELEASE

Step2: Create a new Maven project by entering a Group Id and Artifact Id:

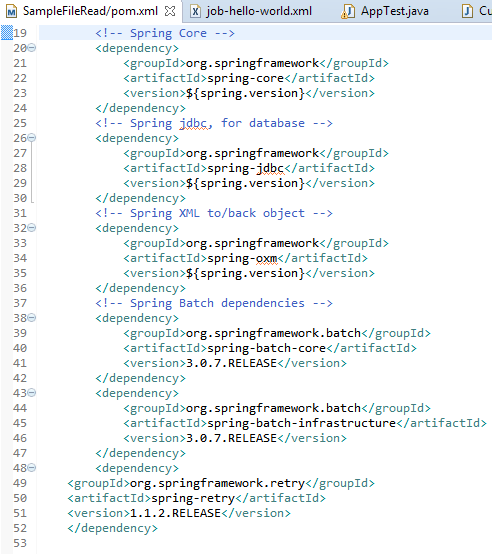




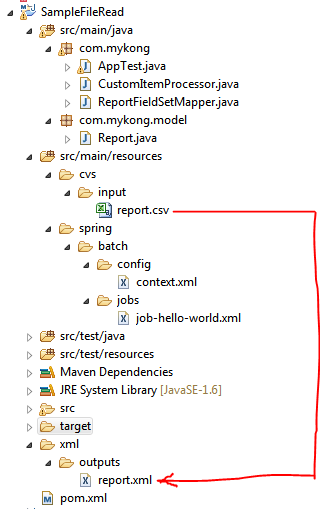


**Step3**: Add spring core and batch dependencies to the POM and run as maven install.

Project dependencies:

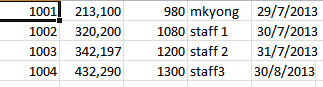


**Step4**: Project Structure:

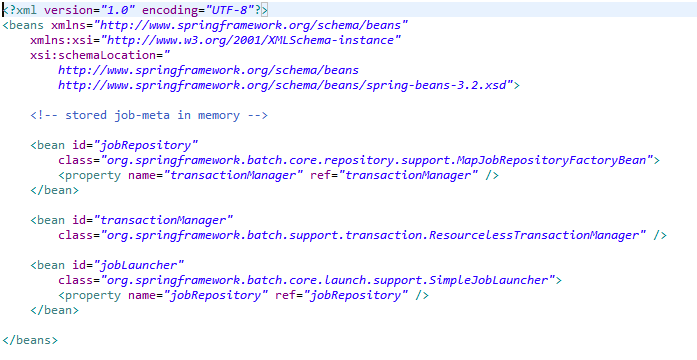


**Step5:**

**Sample report.csv file:**

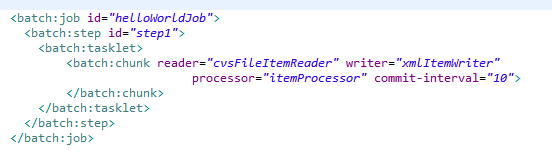


**Step 6:** In Context.xml, define job repository , job launchers and transaction managers and data source.



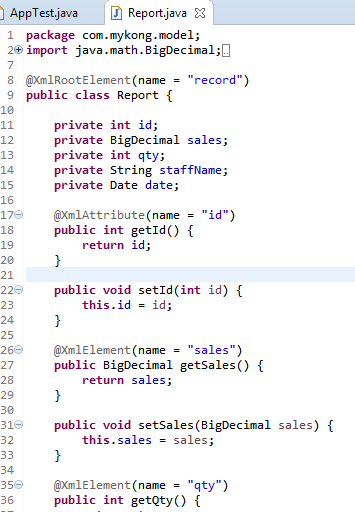
**Step7: In job-hello-world.xml, define the job , steps, item readers and item writers.**

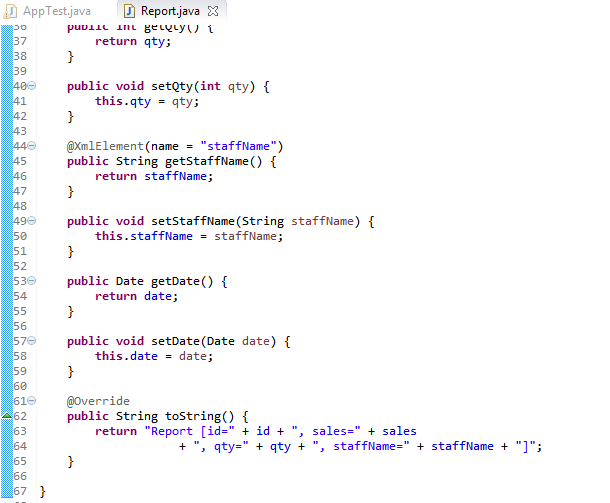
**PFB Sample Job configuration:**



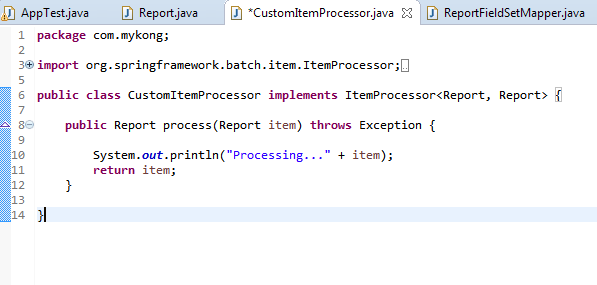
****

**Report.java:**

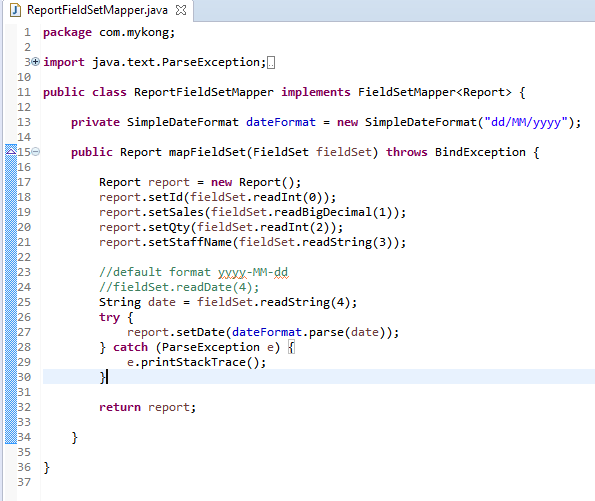




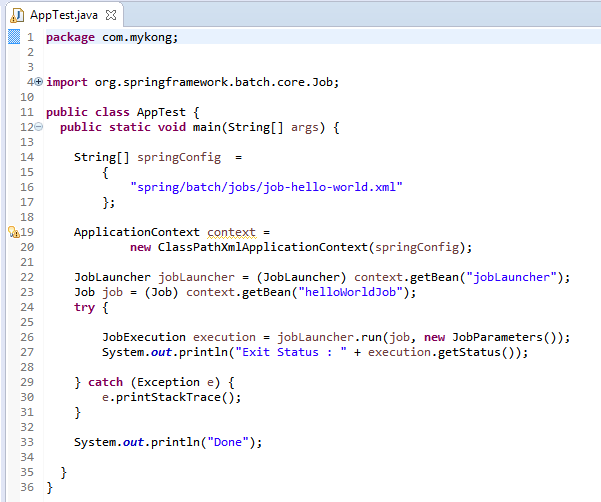
**Step9: CustomItemProcessor.java - (We can add any processing logic here- if we want to modify the input data-we can write it here)**



**Step 10: ReportFieldSetMapper.java: (To map the fileds of csv file with the modal object)**



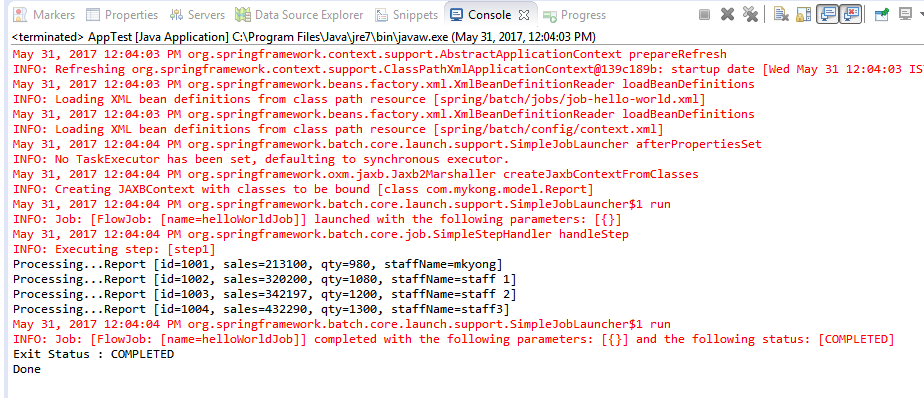
**Step8: In App test.java, execute the job with Job Launcher and parameters. PFB Sample:**



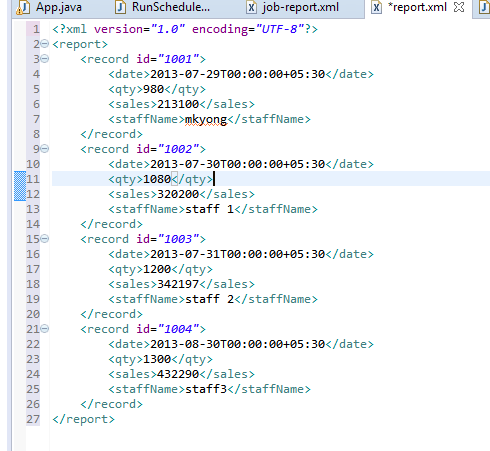
**Step9:**

**Run the AppTest.java as a java Application. Once the job is success we can see the report. XML file with CSV file content**

**Console Output:**



**Report.xml:**



**Attached the source code: Please refer SampleFileRead.zip**

**Job Scheduling:** If we want to run a series of tasks(Job) for every 5 seconds instead of manually running it for every 10 seconds, we can schedule the jobs to run it for every 10 seconds.

Spring 3.0 provides task scheduler to schedule the jobs.

For this we need to add the Task schedulers in Job configuration file , It’s part of the Spring-Core, no need to declare an extra dependency.

<!-- Run every 5 seconds -->

<task:scheduled-tasks>

<task:scheduled ref=*"runScheduler"* **method**=*"run"* **cron**=*"\*/5 \* \* \* \* \*"* />

</task:scheduled-tasks>

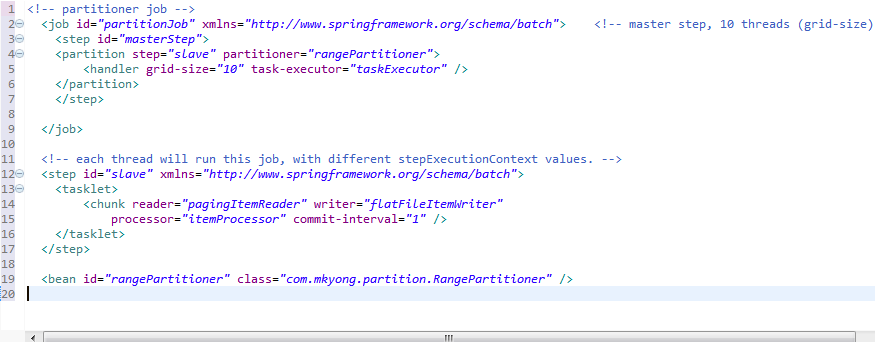
Here The TaskScheduler will schedule to run the bean runScheduler. Here in cron attribute we will define the timeframe.

**Attached the sample program which displays the file contents for every 60 seconds**.

Please refer SampleJobScheduler.zip.

**Multithreading :**

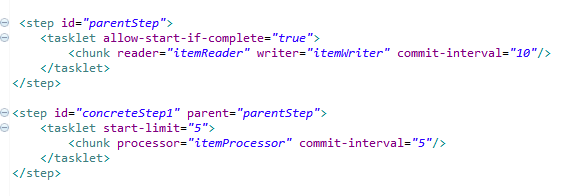
Spring Batch’s multithreaded step concept allows a batch job to use Spring’s [TaskExecutor](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/core/task/TaskExecutor.html) abstraction to execute each chunk in its own thread. a step in a job can be configured to perform within a threadpool, processing each chunk independently. As chunks are processed, Spring Batch keeps track of what is done accordingly. If an error occurs in any one of the threads, the job’s processing is rolled back or terminated per the regular Spring Batch functionality.



**Inheriting a Step:**

Similar to java class inheritance, we can inherit the steps. If a group of Steps share similar configurations, to define a "parent" Step from which the concrete Steps may inherit properties. the "child" Step will combine its elements and attributes with the parent's. The child will also override any of the parent's Steps.

Example:



In above example, the Step "concreteStep1" will inherit from "parentStep". It will be instantiated with 'itemReader', 'itemProcessor', 'itemWriter', startLimit=5, and allowStartIfComplete=true. Additionally, the commitInterval will be '5' since it is overridden by the "concreteStep1":.

**Abstract Step:**

Sometimes it may be necessary to define a parent Step that is not a complete Step configuration. If, for instance, the reader, writer, and tasklet attributes are left off of a Step configuration, then initialization will fail. If a parent must be defined without these properties, then the "abstract" attribute should be used. An "abstract" Step will not be instantiated; it is used only for extending.

In the following example, the Step "abstractParentStep" would not instantiate if it were not declared to be abstract. The Step "concreteStep2" will have 'itemReader', 'itemWriter', and commitInterval=10.



**Restarting a Job:**

The launching of a Job is considered to be a 'restart' if a Job Execution already exists for the particular Job Instance. Ideally, all jobs should be able to start up where they left off, but there are scenarios where this is not possible.

**It is entirely up to the developer to ensure that a new** Job Instance **is created in this scenario**. However, Spring Batch does provide some help. If a Jobs hould never be restarted, but should always be run as part of a new Job Instance, then the restartable property may be set to 'false':

<job id="helloworldJob" **restartable="false"**>

...

</job>

Restarting a Job that is not restartable will cause a JobRestartException to be thrown.

**JOB Listeners**: Intercepting job execution- to notify various events in job life ccyel can be done by calling Job Listeners at the appropriate time. These listeners need to be added for the Job . PFB sample**:**

### We need to override the beforeJob and AfterJob methods of JobexecutionListener.

<job id="sampleJob">

**<listeners>**

**<listener ref="sampleListener"/>**

**</listeners>**

</job>

### Configuring a Step for Restart:

#### Setting a StartLimit : If we want to control the number of times a Step may be started , we have to set the StartLimit parameter. Defaut value for the start limit is Interger.MAX\_VALUE. Suppose if we want the step has to be executed only once, then

<step id="step1">

<tasklet start-limit="1">

<chunk reader="itemReader" writer="itemWriter" commit-interval="10"/>

</tasklet>

</step>

#### Restarting a completed step : In the case of a restartable job, there may be one or more steps that should always be run, regardless of whether or not they were successful the first time. An example might be a validation step, or a Step that cleans up resources before processing. During normal processing of a restarted job, any step with a status of 'COMPLETED', meaning it has already been completed successfully, will be skipped. Setting allow-start-if-complete to "true" overrides this so that the step will always run:

<step id="step1">

<tasklet allow-start-if-complete="true">

<chunk reader="itemReader" writer="itemWriter" commit-interval="10"/>

</tasklet>

</step>

### Configuring Skip Logic

While processing any step, if any errors encounters, it will results a step failure. To avoid step failure, we can skip it . Skipping a step decision depends on the data . For this, we have to add Skippable exception classes in step configuration by adding the exception which need to be included and excluded.

<step id="step1">

<tasklet>

<chunk reader="flatFileItemReader" writer="itemWriter"

commit-interval="10" **skip-limit="10"**>

**<skippable-exception-classes>**

**<include class="org.springframework.batch.item.file.FlatFileParseException"/>**

**</skippable-exception-classes>**

</chunk>

</tasklet>

</step>

### Configuring Retry Logic: If we encounter any of the errors like DeadLock scenarios, we need to retry the job. To retry it add retryable exception classes.

**<step>**

**<retryable-exception-classes>**

**<include class="org.springframework.dao.DeadlockLoserDataAccessException"/>**

**</retryable-exception-classes>**

**</step>**

### Controlling Rollback : By default, regardless of retry or skip, any exceptions thrown from the ItemWriter will cause the transaction controlled by the Step to rollback. If skip is configured as described above, exceptions thrown from the ItemReader will not cause a rollback. However, there are many scenarios in which exceptions thrown from the ItemWriter should not cause a rollback because no action has taken place to invalidate the transaction. For this reason, the Step can be configured with a list of exceptions that should not cause rollback.

<no-rollback-exception-classes>

<include class="org.springframework.batch.item.validator.ValidationException"/>

</no-rollback-exception-classes>

#### Transactional Readers : By default, Step buffers the reader input. So that in case of rollback, items don’t need to be re-read from the reader. In some scenarios like where the reader is built on top of a transactional resource, such as JMS queue. In this case, since the queue is tied to the transaction that is rolled back, messages that have been pulled from the queue will be put back on.

For this reason the step can be configured to not buffer the items:

<step id=*"step1"*>

<tasklet>

<chunk reader=*"itemReader"* writer=*"itemWriter"* commit-interval=*"2"*

is-reader-transactional-queue=*"true"*/>

</tasklet>

</step>

**Transaction Attributes :**

Transaction attributes can be used to control the isolation, propagation, and timeout settings.

<step id=*"step1"*>

<tasklet>

<chunk reader=*"itemReader"* writer=*"itemWriter"* commit-interval=*"2"*/>

<transaction-attributes isolation=*"DEFAULT"*

propagation=*"REQUIRED"*

timeout=*"30"*/>

</tasklet>

</step>

### 