**Introduction To spring batch**

Many applications within the enterprise domain require bulk processing to perform business operations in mission critical environments. These business operations include automated, complex processing of large volumes of information that is most efficiently processed without user interaction.

Batch processing is used to process billions of transactions every day for enterprises.

It is intended to work in conjunction with a scheduler, not replace a scheduler. Conclusion: a scheduler will set and then call a spring batch job , so scheduler comes above in the hierarchy .

It also provides more advance technical services and features that will enable extremely high-volume and high performance batch jobs though optimization and partitioning techniques.

A typical batch program generally reads a large number of records from a database, file, or queue, processes the data in some fashion, and then writes back data in a modified form. Spring Batch automates this basic batch iteration, providing the capability to process similar transactions as a set, typically in an offline environment without any user interaction. Batch jobs are part of most IT projects and Spring Batch is the only open source framework that provides a robust, enterprise-scale solution.

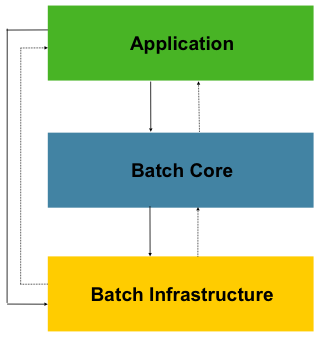
Business Scenarios

* Commit batch process periodically
* Concurrent batch processing: parallel processing of a job
* Staged, enterprise message-driven processing
* Massively parallel batch processing
* Manual or scheduled restart after failure
* Sequential processing of dependent steps (with extensions to workflow-driven batches)
* Partial processing: skip records (e.g. on rollback)
* Whole-batch transaction: for cases with a small batch size or existing stored procedures/scripts

Technical Objectives

* 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.
* Provide 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.
* Provide a simple deployment model, with the architecture JARs completely separate from the application, built using Maven.
* xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Architecture



This layered architecture highlights three major high level components: Application, Core, and Infrastructure. The application contains all batch jobs and custom code written by developers using Spring Batch. The Batch Core contains the core runtime classes necessary to launch and control a batch job. It includes things such as aJobLauncher, Job, and Step implementations. Both Application and Core are built on top of a common infrastructure. This infrastructure contains common readers and writers, and services such as the RetryTemplate, which are used both by application developers(ItemReader and ItemWriter) and the core framework itself. (retry)

## Batch Processing Strategies

* Conversion Applications: For each type of file supplied by or generated to an external system, a conversion application will need to be created to convert the transaction records supplied into a standard format required for processing. This type of batch application can partly or entirely consist of translation utility modules (see Basic Batch Services).
* Validation Applications: Validation applications ensure that all input/output records are correct and consistent. Validation is typically based on file headers and trailers, checksums and validation algorithms as well as record level cross-checks.
* Extract Applications: An application that reads a set of records from a database or input file, selects records based on predefined rules, and writes the records to an output file.
* Extract/Update Applications: An application that reads records from a database or an input file, and makes changes to a database or an output file driven by the data found in each input record.
* Processing and Updating Applications: An application that performs processing on input transactions from an extract or a validation application. The processing will usually involve reading a database to obtain data required for processing, potentially updating the database and creating records for output processing.
* Output/Format Applications: Applications reading an input file, restructures data from this record according to a standard format, and produces an output file for printing or transmission to another program or system.

**utility steps**

* Sort - A Program that reads an input file and produces an output file where records have been re-sequenced according to a sort key field in the records. Sorts are usually performed by standard system utilities.
* Split - A program that reads a single input file, and writes each record to one of several output files based on a field value. Splits can be tailored or performed by parameter-driven standard system utilities.
* Merge - A program that reads records from multiple input files and produces one output file with combined data from the input files. Merges can be tailored or performed by parameter-driven standard system utilities.

**Batch applications can additionally be categorized by their input source:**

* **Database-driven applications** are driven by rows or values retrieved from the database.
* **File-driven applications** are driven by records or values retrieved from a file.
* **Message-driven applications** are driven by messages retrieved from a message queue.

Xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

**Partitioning** Using partitioning allows multiple versions of large batch applications to run concurrently. The purpose of this is to reduce the elapsed time required to process long batch jobs. Processes which can be successfully partitioned are those where the input file can be split and/or the main database tables partitioned to allow the application to run against different sets of data.

Fixed and Even Break-Up of Record Set

Breakup by a Key Column………………… etc

**Parameter Passing and Validation**

The architecture should perform all tasks associated with running the application in a partitioned mode including:

* Retrieve partition parameters before application start-up
* Validate partition parameters before application start-up
* Pass parameters to application at start-up

The validation should include checks to ensure that:

* the application has sufficient partitions to cover the whole data range
* there are no gaps between partitions

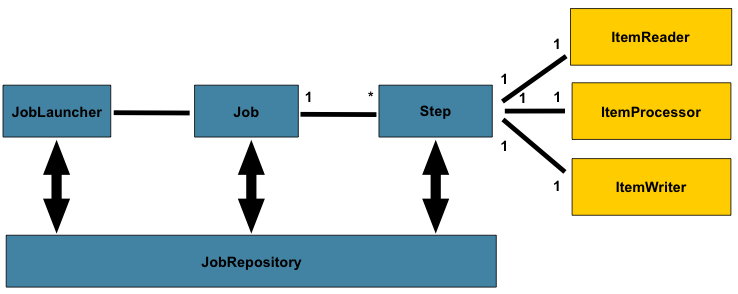
If the database is partitioned, some additional validation may be necessary to ensure that a single partition does not span database partitions.

Also the architecture should take into consideration the consolidation of partitions. Key questions include:

Spring Batch now supports being run on Java 8. It will still execute on Java 6 or higher as well.

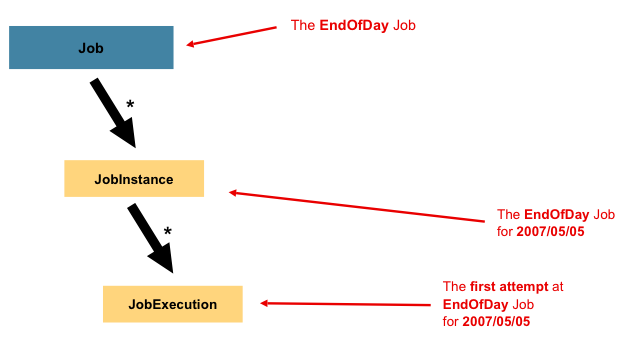
## JobScope Support

The Spring scope "step" used in Spring Batch has had a pivotal role in batch applications, providing late binding functionality for a long time now. With the 3.0 release Spring Batch now supports a "job" scope. This new scope allows for the delayed construction of objects until a Job is actually launched as well as providing a facility for new instances for each execution of a job.



A Job has one to many steps, which has exactly one ItemReader, ItemProcessor, and ItemWriter. A job needs to be launched (JobLauncher), and meta data about the currently running process needs to be stored (JobRepository).

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. This configuration may be referred to as the "job configuration". However, Job is just the top of an overall hierarchy:



In Spring Batch, a Job is simply a container for Steps. It combines multiple steps that belong logically together in a flow

Eg.

<job id="footballJob">

<step id="playerload" next="gameLoad"/>

<step id="gameLoad" next="playerSummarization"/>

<step id="playerSummarization"/>

</job>

**JOB INSTANCE**

The definition of a JobInstance has absolutely no bearing on the data the will be loaded. It is entirely up to the ItemReader implementation used to determine how data will be loaded.

Using a new JobInstance will mean 'start from the beginning' and using an existing instance will generally mean 'start from where you left off'.

### JobParameters

how is one JobInstance distinguished from another?" The answer is:JobParameters

Thus, the contract can be defined as: JobInstance = Job + identifyingJobParameters. This allows a developer to effectively control how a JobInstance is defined, since they control what parameters are passed in.

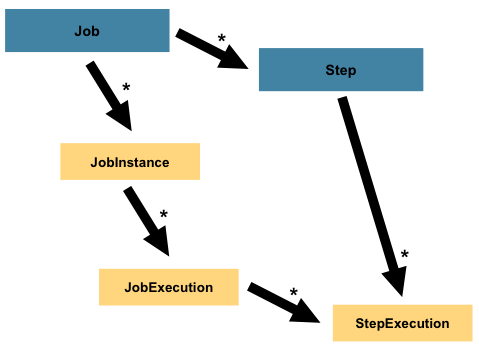
### JobExecution

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. 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 aJobInstance for 01-01-2008 that failed the first time it was run. If it is run again with the same identifying job parameters as the first run (01-01-2008), a newJobExecution will be created. However, there will still be only one JobInstance.

## 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. AStep contains all of the information necessary to define and control the actual batch processing.

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



### 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.

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 will contain an ExecutionContext, which contains any data a developer needs persisted across batch runs, such as statistics or state information needed to restart.

## ExecutionContext

An ExecutionContext represents a collection of key/value pairs that are persisted and controlled by the framework in order to allow developers a place to store persistent state that is scoped to a StepExecution or JobExecution. For those familiar with Quartz, it is very similar to JobDataMap. The best usage example is to facilitate restart. Using flat file input as an example, while processing individual lines, the framework periodically persists the ExecutionContext at commit points. This allows the ItemReader to store its state in case a fatal error occurs during the run, or even if the power goes out. All that is needed is to put the current number of lines read into the context, and the framework will do the rest:

executionContext.putLong(getKey(LINES\_READ\_COUNT), reader.getPosition());

## JobRepository

JobRepository is the persistence mechanism for all of the Stereotypes mentioned above. It provides CRUD operations for JobLauncher, Job, and Stepimplementations. When a Job is first launched, a JobExecution is obtained from the repository, and during the course of execution StepExecution andJobExecution implementations are persisted by passing them to the repository:

<job-repository id="jobRepository"/>

## JobLauncher

JobLauncher represents a simple interface for launching a Job with a given set of JobParameters:

**public** **interface** JobLauncher {

**public** JobExecution run(Job job, JobParameters jobParameters)

**throws** JobExecutionAlreadyRunningException, JobRestartException;

}

It is expected that implementations will obtain a valid JobExecution from the JobRepository and execute the Job.

ItemReader is a basic interface for generic input operations:

**public** **interface** ItemReader<T> {

T read() **throws** Exception, UnexpectedInputException, ParseException;

}

The read method defines the most essential contract of the ItemReader; calling it returns one Item or null if no more items are left. An item might represent a line in a file, a row in a database, or an element in an XML file. It is generally expected that these will be mapped to a usable domain object (i.e. Trade, Foo, etc) but there is no requirement in the contract to do so.

It is expected that implementations of the ItemReader interface will be forward only

ItemWriter is similar in functionality to an ItemReader, but with inverse operations. Resources still need to be located, opened and closed but they differ in that anItemWriter writes out, rather than reading in. In the case of databases or queues these may be inserts, updates, or sends. The format of the serialization of the output is specific to each batch job.

As with ItemReader, ItemWriter is a fairly generic interface:

**public** **interface** ItemWriter<T> {

**void** write(List<? **extends** T> items) **throws** Exception;

}

As with read on ItemReader, write provides the basic contract of ItemWriter; it will attempt to write out the list of items passed in as long as it is open. Because it is generally expected that items will be 'batched' together into a chunk and then output, the interface accepts a list of items, rather than an item by itself. After writing out the list, any flushing that may be necessary can be performed before returning from the write method. For example, if writing to a Hibernate DAO, multiple calls to write can be made, one for each item. The writer can then call close on the hibernate Session before returning.

## ItemProcessor

The ItemReader and ItemWriter interfaces are both very useful for their specific tasks, but what if you want to insert business logic before writing? One option for both reading and writing is to use the composite pattern: create an ItemWriter that contains another ItemWriter, or an ItemReader that contains anotherItemReader

**Exploring Spring batch AUTODATA project : Portfolio manager**

**Inheriting from a Parent Job**

If a group of Jobs share similar, but not identical, configurations, then it may be helpful to define a "parent" Job from which the concrete Jobs may inherit properties. Similar to class inheritance in Java, the "child" Job will combine its elements and attributes with the parent's.

**Eg.**

<batch:job id=*"rootJob"* abstract=*"true"*>

<batch:listeners>

<batch:listener ref=*"jobEmailListenerAlways"*/>

<batch:listener ref=*"analyticsJob"*/>

</batch:listeners>

</batch:job>

<batch:job id=*"ETLFeedsProcessingJob"* parent=*"rootJob"*>

#### Merging Lists

Some of the configurable elements on Steps are lists; the <listeners/> element, for instance. If both the parent and child Steps declare a <listeners/> element, then the child's list will override the parent's. In order to allow a child to add additional listeners to the list defined by the parent, every list element has a "merge" attribute. If the element specifies that merge="true", then the child's list will be combined with the parent's instead of overriding it.

In the following example, the Step "concreteStep3" will be created will two listeners: listenerOne and listenerTwo:

<step id="listenersParentStep" abstract="true">

<listeners>

<listener ref="listenerOne"/>

<listeners>

</step>

<step id="concreteStep3" parent="listenersParentStep">

<tasklet>

<chunk reader="itemReader" writer="itemWriter" commit-interval="5"/>

</tasklet>

<listeners merge="true">

<listener ref="listenerTwo"/>

<listeners>

</step>

What are listeners in Spring batch ??

**Intercepting Job Execution** :{ we are using it in our project }

During the course of the execution of a Job, it may be useful to be notified of various events in its lifecycle so that custom code may be executed. The SimpleJoballows for this by calling a JobListener at the appropriate time:

**public** **interface** JobExecutionListener {

**void** beforeJob(JobExecution jobExecution);

**void** afterJob(JobExecution jobExecution);

}

It should be noted that afterJob will be called regardless of the success or failure of the Job. If success or failure needs to be determined it can be obtained from theJobExecution:

The annotations corresponding to this interface are:

* @BeforeJob
* @AfterJob

### Intercepting Step Execution :{ we are using it in our project }

Any class that implements one of the extensions of StepListener (but not that interface itself since it is empty) can be applied to a step via the listeners element. The listeners element is valid inside a step, tasklet or chunk declaration. It is recommended that you declare the listeners at the level which its function applies, or if it is multi-featured (e.g. StepExecutionListener and ItemReadListener) then declare it at the most granular level that it applies (chunk in the example given).

<step id="step1">

<tasklet>

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

<listeners>

<listener ref="chunkListener"/>

</listeners>

</tasklet>

</step>

#### StepExecutionListener

StepExecutionListener represents the most generic listener for Step execution. It allows for notification before a Step is started and after it has ends, whether it ended normally or failed:

**public** **interface** StepExecutionListener **extends** StepListener {

**void** beforeStep(StepExecution stepExecution);

ExitStatus afterStep(StepExecution stepExecution);

}

The annotations corresponding to this interface are:

* @BeforeStep
* @AfterStep

### Tasklet and its use

### Many batch jobs contain steps that must be done before the main processing begins in order to set up various resources or after processing has completed to cleanup those resources. In the case of a job that works heavily with files, it is often necessary to delete certain files locally after they have been uploaded successfully to another location

### In Spring batch, the Tasklet is an interface, which will be called to perform a single task only, like clean or set up resources before or after any step execution. In this example, we will show you how to use Tasklet to clean up the resource (folders) after a batch job is completed

### X………………………… OVER …………………………………………………….X

### <batch:next on=*"\*"* to=*"ddsImport"*/>

The next element specifies a pattern to match and the step to execute next:

The "on" attribute of a transition element uses a simple pattern-matching scheme to match the ExitStatus that results from the execution of the Step. Only two special characters are allowed in the pattern:

* "\*" will zero or more characters
* "?" will match exactly one character

For example, "c\*t" will match "cat" and "count", while "c?t" will match "cat" but not "count".

While there is no limit to the number of transition elements on a Step, if the Step's execution results in an ExitStatus that is not covered by an element, then the framework will throw an exception and the Job will fail.

#### Batch Status vs. Exit Status

When configuring a Job for conditional flow, it is important to understand the difference between BatchStatus and ExitStatus. BatchStatus is an enumeration that is a property of both JobExecution and StepExecution and is used by the framework to record the status of a Job or Step. It can be one of the following values: COMPLETED, STARTING, STARTED, STOPPING, STOPPED, FAILED, ABANDONED or UNKNOWN. Most of them are self explanatory: COMPLETED is the status set when a step or job has completed successfully, FAILED is set when it fails, and so on

At first glance, it would appear that the 'on' attribute references the BatchStatus of the Step to which it belongs. However, it actually references the ExitStatus of the Step. As the name implies, ExitStatus represents the status of a Step after it finishes execution. More specifically, the 'next' element above references the exit code of the ExitStatus. To write it in English, it says: "go to stepB if the exit code is FAILED". By default, the exit code is always the same as the BatchStatus for the Step, which is why the entry above works

Eg.

<step id="step1" parent="s1">

<end on="FAILED" />

<next on="COMPLETED WITH SKIPS" to="errorPrint1" />

<next on="\*" to="step2" />

</step>

The above step has three possibilities:

1. The Step failed, in which case the job should fail.
2. The Step completed successfully.
3. The Step completed successfully, but with an exit code of 'COMPLETED WITH SKIPS'. In this case, a different step should be run to handle the errors.

Now to insert or customize the exit status you need to add a StepExecutionListener  where **public** ExitStatus afterStep(StepExecution stepExecution) {

Method need to be implemented using your logic

### Split Flows

Spring Batch namespace also allows for a job to be configured with parallel flows using the 'split' element. As is seen below, the 'split' element contains one or more 'flow' elements, where entire separate flows can be defined. A 'split' element may also contain any of the previously discussed transition elements such as the 'next' attribute or the 'next', 'end', 'fail', or 'pause' elements.

<split id="split1" next="step4">

<flow>

<step id="step1" parent="s1" next="step2"/>

<step id="step2" parent="s2"/>

</flow>

<flow>

<step id="step3" parent="s3"/>

</flow>

</split>

<step id="step4" parent="s4"/>

**XX…………………………………………………………… over ………………………………………………….……………………………XX**

**MULTITHREADED STEP/ job**

At a high level there are two modes of parallel processing: single process, multi-threaded; and multi-process. These break down into categories as well, as follows:

* Multi-threaded Step (single process)
* Parallel Steps (single process)
* Remote Chunking of Step (multi process)
* Partitioning a Step (single or multi process)

## Multi-threaded Step

The simplest way to start parallel processing is to add a TaskExecutor to your Step configuration, e.g. as an attribute of the tasklet:

<step id="loading">

<tasklet task-executor="taskExecutor">...</tasklet>

</step>

In this example the taskExecutor is a reference to another bean definition, implementing the TaskExecutor interface. TaskExecutor is a standard Spring interface, so consult the Spring User Guide for details of available implementations. The simplest multi-threaded TaskExecutor is a SimpleAsyncTaskExecutor.

In addition to any limits placed by the task executor (e.g. if it is backed by a thread pool), there is a throttle limit in the tasklet configuration which defaults to 4. You may need to increase this to ensure that a thread pool is fully utilized

## Parallel Steps {used in our project }

As long as the application logic that needs to be parallelized can be split into distinct responsibilities, and assigned to individual steps then it can be parallelized in a single process. Parallel Step execution is easy to configure and use, for example, to execute steps (step1,step2) in parallel with step3, you could configure a flow like this:

<job id="job1">

<split id="split1" task-executor="taskExecutor" next="step4">

<flow>

<step id="step1" parent="s1" next="step2"/>

<step id="step2" parent="s2"/>

</flow>

<flow>

<step id="step3" parent="s3"/>

</flow>

</split>

<step id="step4" parent="s4"/>

</job>

<beans:bean id="taskExecutor" class="org.spr...SimpleAsyncTaskExecutor"/>

The configurable "task-executor" attribute is used to specify which TaskExecutor implementation should be used to execute the individual flows. The default isSyncTaskExecutor, but an asynchronous TaskExecutor is required to run the steps in parallel. Note that the job will ensure that every flow in the split completes before aggregating the exit statuses and transitioning.

#### **Chunk Processing**

Chunk-oriented processing is the most commonly encountered operation style in which the processing happens in certain ‘chunks’ or blocks of data defined by a transaction boundary. That is, the [itemReader](http://docs.spring.io/spring-batch/trunk/reference/html/readersAndWriters.html" \l "itemReader" \t "_blank) reads a piece of data which are then fed to the [itemProcessor](http://docs.spring.io/spring-batch/trunk/reference/html/readersAndWriters.html" \l "itemProcessor" \t "_blank) and aggregated till the transaction limit is reached. Once it does, the aggregated data is passed over to the [itemWriter](http://docs.spring.io/spring-batch/trunk/reference/html/readersAndWriters.html" \l "itemWriter" \t "_blank) to write out the data. The size of the chunk is specified by the ‘commit-interval’ attribute as shown in the snippet below.  
Step

|  |  |
| --- | --- |
| 1 | <step id="springBatchCsvToXmlProcessor"> |
| 2 | <chunk reader="itemReader" writer="xmlWriter" commit-interval="10"></chunk> |

|  |  |
| --- | --- |
| 3 | </step> |

The following diagram from the Spring Documentation summarizes the operation pretty well.

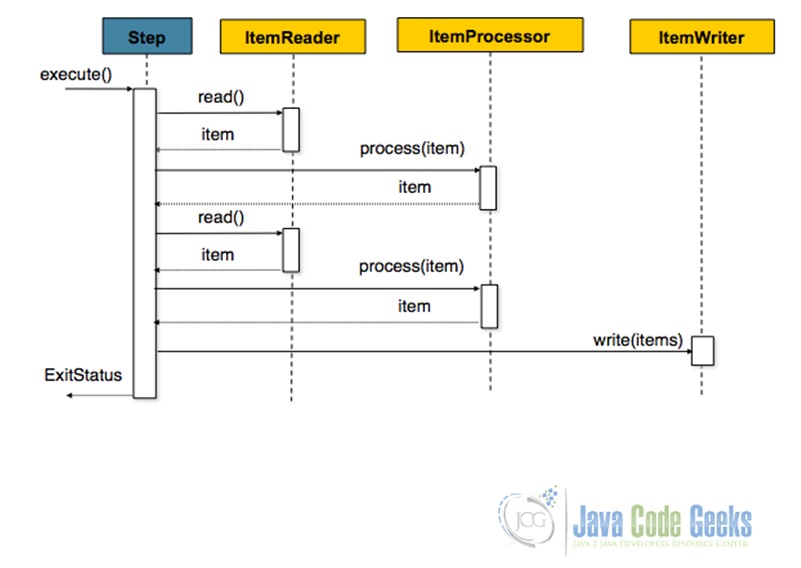
[](http://examples.javacodegeeks.com/wp-content/uploads/2015/06/chunkProcessing.jpg)

Fig.2 Chunk-Oriented Processing

### Commit Interval

With a commit-interval of 1, it will commit after writing each individual item.

**commit-interval="10"**/>

10 items will be processed within each transaction. At the beginning of processing a transaction is begun, and each time read is called on theItemReader, a counter is incremented. When it reaches 10, the list of aggregated items is passed to the ItemWriter, and the transaction will be committed.