## Spring Batch

## Usage Scenarios

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

**1.3 Spring Batch Architecture**

Spring Batch is designed with extensibility and a diverse group of end users in mind. The figure below shows a sketch of the layered architecture that supports the extensibility and ease of use for end-user developers.

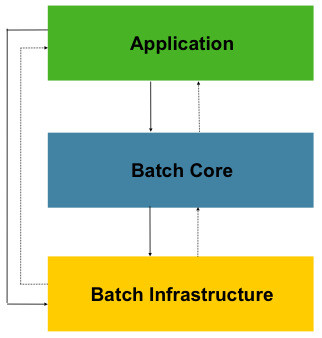


Figure 1.1: Spring Batch Layered Architecture

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 a JobLauncher, 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)

**Concurrent batch / on-line processing** Batch applications processing data that can simultaneously be updated by on-line users, should not lock any data (either in the database or in files) which could be required by on-line users for more than a few seconds. Also updates should be committed to the database at the end of every few transaction. This minimizes the portion of data that is unavailable to other processes and the elapsed time the data is unavailable.

Another option to minimize physical locking is to have a logical row-level locking implemented using either an Optimistic Locking Pattern or a Pessimistic Locking Pattern.

* Optimistic locking assumes a low likelihood of record contention. It typically means inserting a timestamp column in each database table used concurrently by both batch and on-line processing. When an application fetches a row for processing, it also fetches the timestamp. As the application then tries to update the processed row, the update uses the original timestamp in the WHERE clause. If the timestamp matches, the data and the timestamp will be updated successfully. If the timestamp does not match, this indicates that another application has updated the same row between the fetch and the update attempt and therefore the update cannot be performed.
* Pessimistic locking is any locking strategy that assumes there is a high likelihood of record contention and therefore either a physical or logical lock needs to be obtained at retrieval time. One type of pessimistic logical locking uses a dedicated lock-column in the database table. When an application retrieves the row for update, it sets a flag in the lock column. With the flag in place, other applications attempting to retrieve the same row will logically fail. When the application that set the flag updates the row, it also clears the flag, enabling the row to be retrieved by other applications. Please note, that the integrity of data must be maintained also between the initial fetch and the setting of the flag, for example by using db locks (e.g., SELECT FOR UPDATE). Note also that this method suffers from the same downside as physical locking except that it is somewhat easier to manage building a time-out mechanism that will get the lock released if the user goes to lunch while the record is locked.

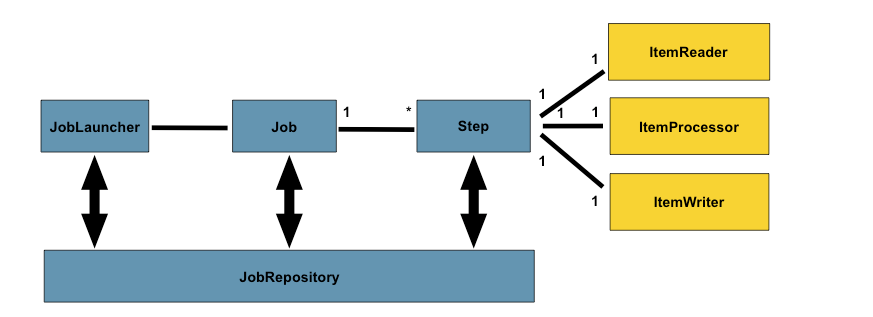
These patterns are not necessarily suitable for batch processing, but they might be used for concurrent batch and on-line processing (e.g. in cases where the database doesn't support row-level locking). **As a general rule, optimistic locking is more suitable for on-line applications, while pessimistic locking is more suitable for batch applications**. Whenever logical locking is used, the same scheme must be used for all applications accessing data entities protected by logical locks.

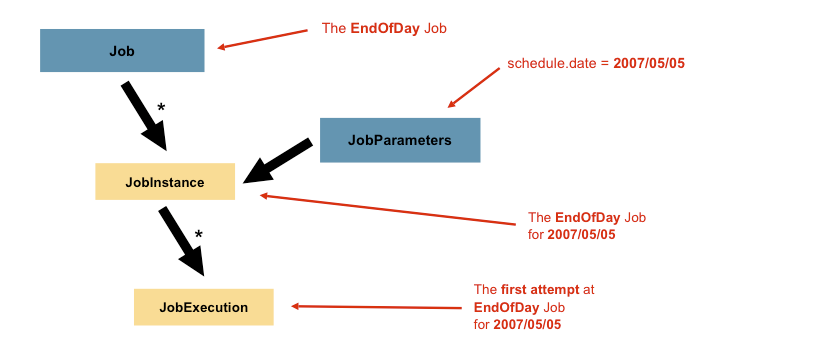
Note that both of these solutions only address locking a single record. Often we may need to lock a logically related group of records. With physical locks, you have to manage these very carefully in order to avoid potential deadlocks. With logical locks, it is usually best to build a logical lock manager that understands the logical record groups you want to protect and can ensure that locks are coherent and non-deadlocking. This logical lock manager usually uses its own tables for lock management, contention reporting, time-out mechanism, etc.

**Parallel Processing** Parallel processing allows multiple batch runs / jobs to run in parallel to minimize the total elapsed batch processing time. This is not a problem as long as the jobs are not sharing the same files, db-tables or index spaces. If they do, this service should be implemented using **partitioned** data

**Partitioning** Using partitioning allows multiple versions of large batch applications to run concurrently.

**Architecture**

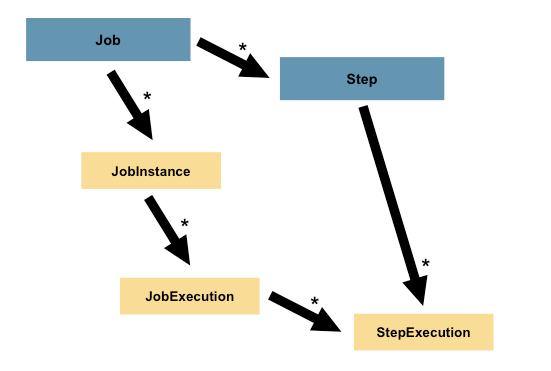




### JobExecution

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

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



### StepExecution

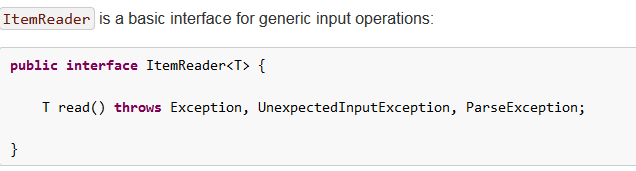
A StepExecution represents a single attempt to execute a Step. A new StepExecution will be created each time a Step is run, similar to JobExecution.

## JobRepository

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

## Item Reader

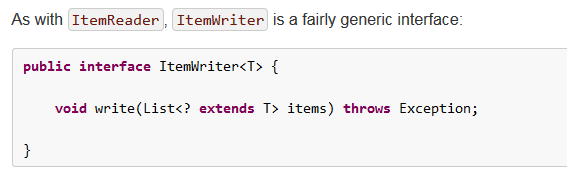
ItemReader is an abstraction that represents the retrieval of input for a Step, one item at a time. When the ItemReader has exhausted the items it can provide, it will indicate this by returning null. More details about the ItemReader



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.

**ItemWriter**

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 an ItemWriter 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.

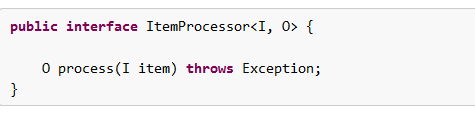


As with read on ItemReader, write provides the basic contract of ItemWriter; it will attempt to write out the list of items passed in as long as it is open. Because it is generally expected that items will be 'batched' together into a chunk and then output, the interface accepts a list of items, rather 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 another ItemReader.

An ItemProcessor is very simple; given one object, transform it and return another. The provided object may or may not be of the same type. The point is that business logic may be applied within process, and is completely up to the developer to create. An ItemProcessor can be wired directly into a step, For example, assuming an ItemReader provides a class of type Foo, and it needs to be converted to type Bar before being written out. An ItemProcessor can be written that performs the conversion:



Customozation

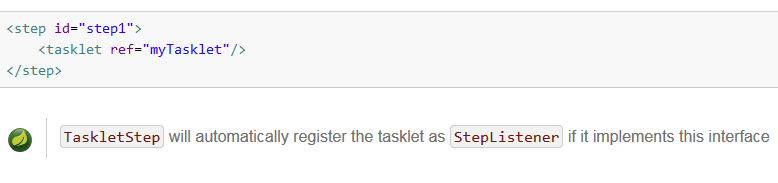
FlatFileItemReader to skip empty files



## TaskletStep

Chunk-oriented processing is not the only way to process in a Step. What if a Step must consist as a simple stored procedure call? You could implement the call as an ItemReader and return null after the procedure finishes, but it is a bit unnatural since there would need to be a no-op ItemWriter. Spring Batch provides the TaskletStep for this scenario.

The Tasklet is a simple interface that has one method, execute, which will be a called repeatedly by the TaskletStep until it either returns RepeatStatus.FINISHED or throws an exception to signal a failure. Each call to the Tasklet is wrapped in a transaction. Tasklet implementors might call a stored procedure, a script, or a simple SQL update statement. To create a TaskletStep, the 'ref' attribute of the <tasklet/> element should reference a bean defining a Tasklet object; no <chunk/> element should be used within the <tasklet/>:



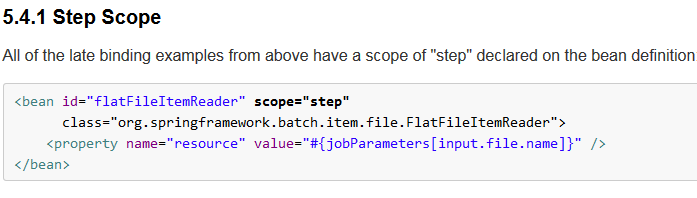
**Chunk oriented and Tasklet processing**

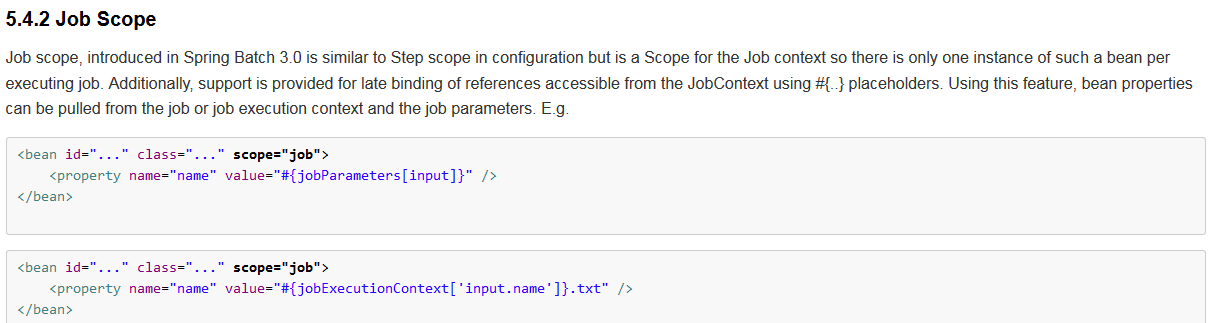
Big Data Sets’ Processing is one of the most important problem in the software world. Spring Batch is a lightweight and robust batch framework to process the data sets.

Spring Batch Framework offers ‘TaskletStep Oriented’ and ‘Chunk Oriented’ processing style. In this article, Chunk Oriented Processing Model is explained. Also, [**TaskletStep Oriented Processing in Spring Batch**](http://www.onlinetechvision.com/?p=658) Article is definitely suggested to investigate how to develop TaskletStep Oriented Processing in Spring Batch.

Chunk Oriented Processing Feature has come with Spring Batch v2.0. It 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 from an **ItemReader**, handed to an **ItemProcessor**, and written. 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.

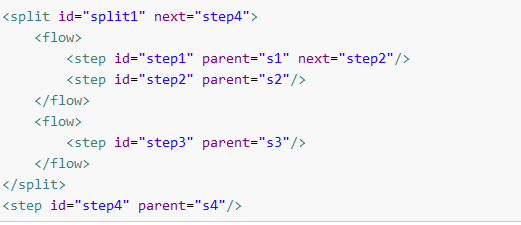
Basically, this feature should be used if at least one data item’ s reading and writing is required. Otherwise, TaskletStep Oriented processing can be used if the data item’ s only reading or writing is required.





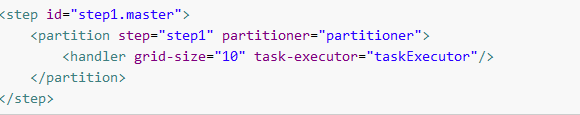
### Split Flows

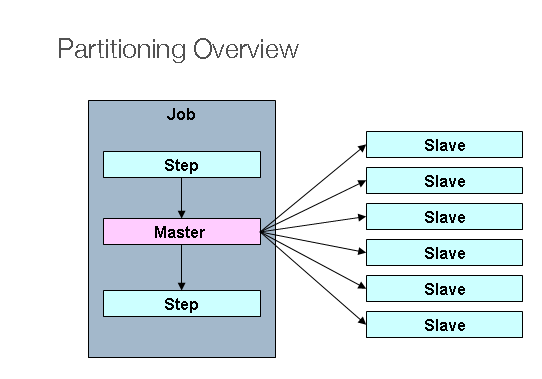
Every scenario described so far has involved a Job that executes its Steps one at a time in a linear fashion. In addition to this typical style, the Spring Batch namespace also allows for a job to be configured with parallel flows using the 'split' element. As is seen below, the 'split' element contains 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.



## Partitioning

Spring Batch also provides an SPI for partitioning a Step execution and executing it remotely. In this case the remote participants are simply Step instances that could just as easily have been configured and used for local processing. Here is a picture of the pattern in action:





The Job is executing on the left hand side as a sequence of Steps, and one of the Steps is labelled as a Master. The Slaves in this picture are all identical instances of a Step, which could in fact take the place of the Master resulting in the same outcome for the Job. The Slaves are typically going to be remote services, but could also be local threads of execution. The messages sent by the Master to the Slaves in this pattern do not need to be durable, or have guaranteed delivery: Spring Batch meta-data in the JobRepository will ensure that each Slave is executed once and only once for each Job execution.

### PartitionHandler

The PartitionHandler is the component that knows about the fabric of the remoting or grid environment. It is able to send StepExecution requests to the remote Steps, wrapped in some fabric-specific format, like a DTO. It does not have to know how to split up the input data, or how to aggregate the result of multiple Step executions. Generally speaking it probably also doesn't need to know about resilience or failover, since those are features of the fabric in many cases, and anyway Spring Batch always provides restartability independent of the fabric: a failed Job can always be restarted and only the failed Steps will be re-executed.

The PartitionHandler interface can have specialized implementations for a variety of fabric types: e.g. simple RMI remoting, EJB remoting, custom web service, JMS, Java Spaces, shared memory grids (like Terracotta or Coherence), grid execution fabrics (like GridGain). Spring Batch does not contain implementations for any proprietary grid or remoting fabrics.

Spring Batch does however provide a useful implementation of PartitionHandler that executes Steps locally in separate threads of execution, using the TaskExecutor strategy from Spring. The implementation is called TaskExecutorPartitionHandler, and it is the default for a step configured with the XML namespace as above. It can also be configured explicitly like this:

