Spring Batch Processing is defined as the Processing of Data without Interaction or interruption

Detail Information:-

1. Spring Batch is an open-source framework for batch processing. Batch Processing in simple terms, refers to running bulk operations that could run for hours on end without needing human intervention. Consider Enterprise level operations that involve say, reading from or writing into or updating millions of database records. Spring Batch provides the framework to have such jobs running with minimum human involvement.

2. It is light-weight, comprehensive, favors POJO-based development approach and comes with all the features that spring offers. Besides, it also exposes a number of classes and APIs that could be exploited say for transaction management, for reading and writing data.

3. Spring Batch provides reusable functions that are essential in processing large volumes of records, including logging/tracing, transaction management, job processing statistics, job restart, skip, resource management, logging, tracing, conversion of data, interfaces, etc. By using these diverse techniques, the framework takes care of the performance and the scalability while processing the records.

4. Spring Batch also provides more advanced technical services and features that will enable extremely high-volume and high performance batch jobs through optimization and partitioning techniques.

5. Spring Batch Application executes a series of jobs (iterative or in parallel), where input data is read, processed and written without any interaction. We are going to see how Spring Batch can help us with this purpose.

6. Normally a batch application can be divided in three main parts:
    - ❖ Reading the data (from a database, file system, etc.)

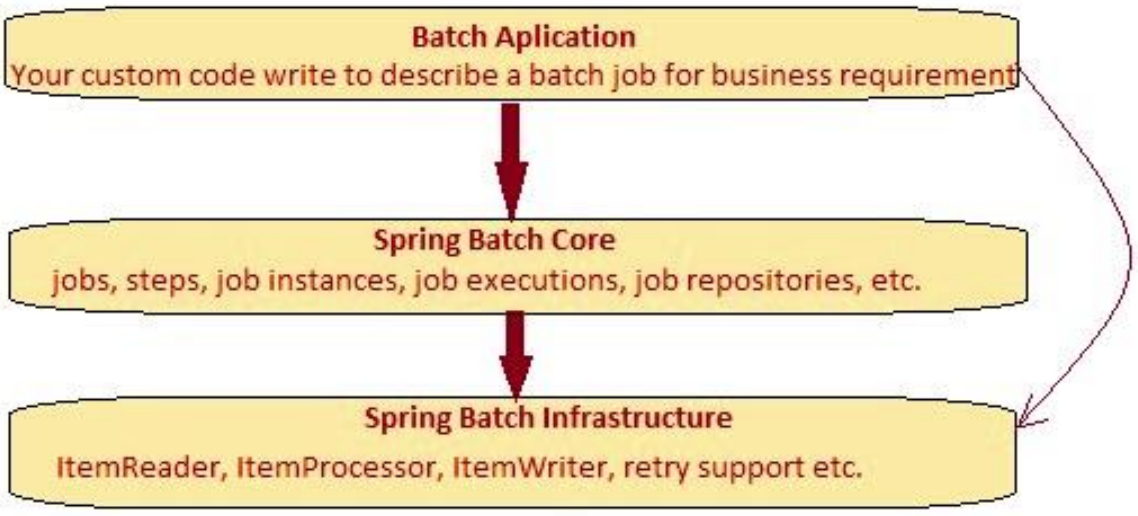
- Processing the data (filtering, grouping, calculating, validating…)
- Writing the data (to a database, reporting, distributing…)

7. Spring Batch contains features and abstractions for automating these basic steps and allowing the application programmers to configure them, repeat them, retry them, stop them, executing them as a single element or grouped (transaction management), etc.

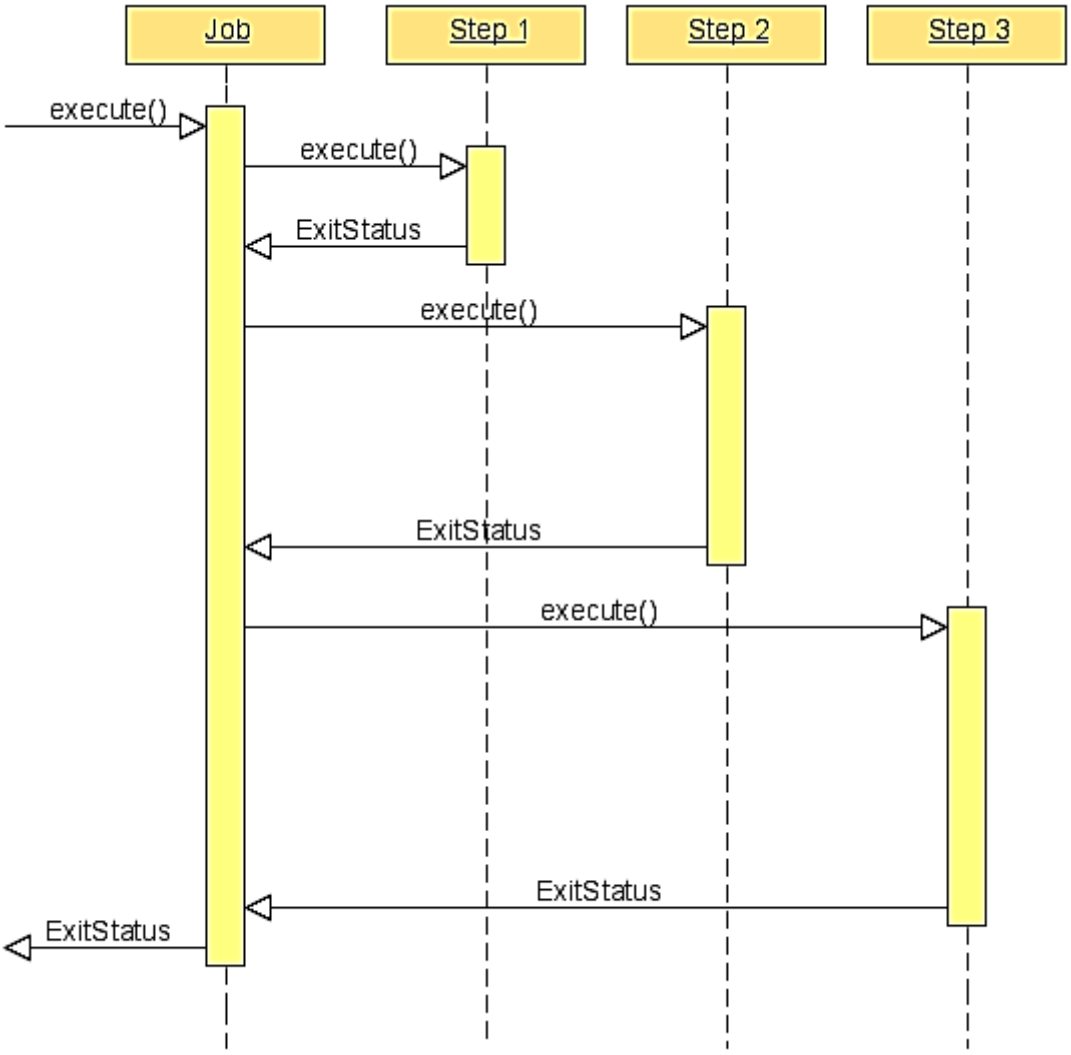
8. It also contains classes and interfaces for the main data formats, industry standards and providers like XML, CSV, SQL, Mongo DB, etc.

9. Examples :-
   Month-end calculations notices or correspondence , periodic application of complex business rules processed repetitively across very large data sets (e.g. insurance benefit determination or rate adjustments)

HIGH LEVEL ARCHITECTURE

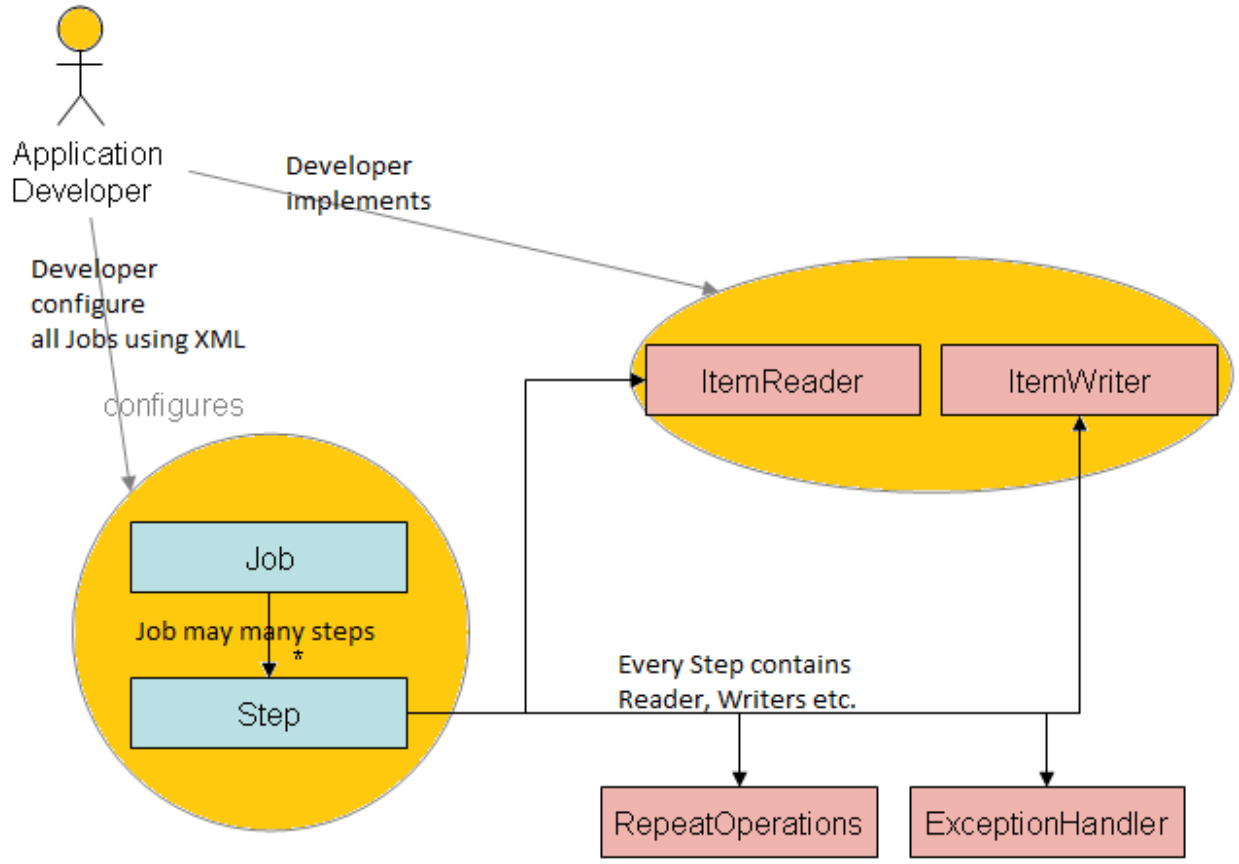


In above figure, the top of the hierarchy is the batch application itself. This is whatever batch processing application you want to write. It depends on the Spring Batch core module, which primarily provides a runtime environment for your batch jobs. Both the batch app and the core module in turn depend upon an infrastructure module that provides classes useful for both building and running batch apps.



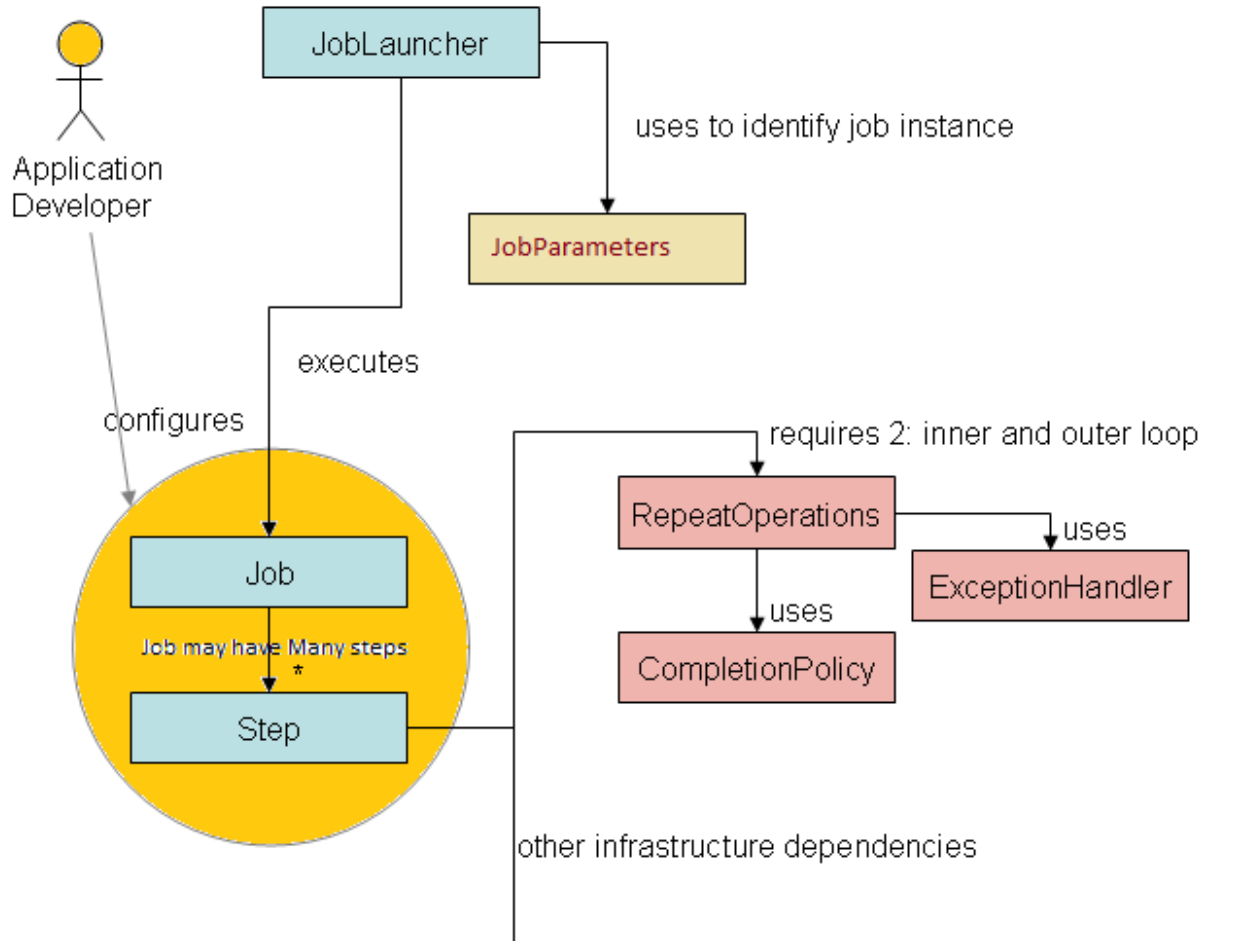
As we see in above figure a hypothetical three-step job, though obviously a job can have arbitrarily many steps. The steps are typically sequential, though as of Spring Batch 2.0 it's possible to define conditional flows (e.g., execute step 2 if step 1 succeeds; otherwise execute step 3). Within any given step, the basic process is as follows: read a bunch of "items" (e.g., database rows, XML elements, and lines in a flat file— whatever), process them, and write them out somewhere to make it convenient for subsequent steps to work with the result.

Overview of the Spring Batch Core - The Spring Batch Core Domain consists of an API for launching, monitoring and managing batch jobs.



The figure above shows the central parts of the core domain and its main touch points with the batch application developer (*Job and Step*). To launch a job there is a *JobLauncher* interface that can be used to simplify the launching for dumb clients like JMX or a command line.

A Job is composed of a list of Steps, each of which is executed in turn by the Job. The *Step* is a central strategy in the Spring Batch Core. Implementations of Step are responsible for sharing the work out, but in ways that the configuration doesn't need to be aware of. For instance, the same or very similar Step configuration might be used in a simple in-process sequential executor, or in a multi-threaded implementation, or one that delegates to remote calls to a distributed system.

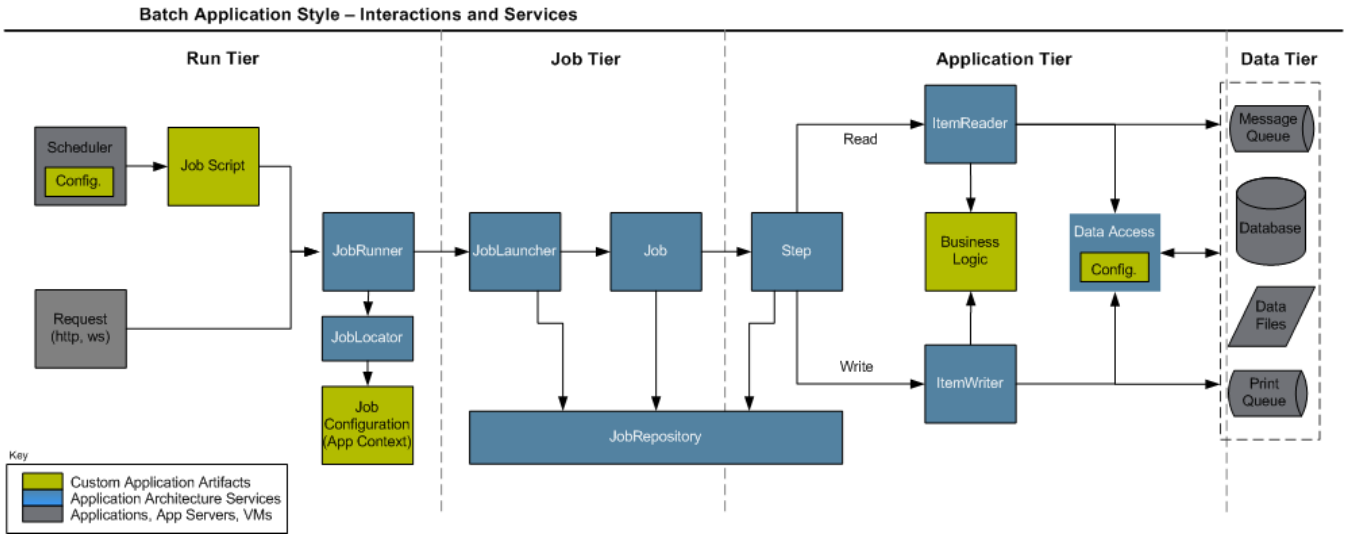


A Job can be re-used to create multiple job instances and this is reflected in the figure above showing an extended picture of the core domain. When a Job is launched it first checks to see if a job with the same *JobParameters* was already executed. We expect one of the following outcomes, depending on the Job:

- If the job was not previously launched then it can be created and executed. A new *JobInstance* is created and stored in a repository (usually a database). A new *JobExecution* is also created to track the progress of this particular execution.
- If the job was previously launched and failed the Job is responsible for indicating whether it believes it is restartable (is a restart legal and expected). There is a flag for this purpose on the Job. If the there was a previous failure - maybe the operator has fixed some bad input and wants to run it again - then we might want to restart the previous job.
- If the job was previously launched with the same *JobParameters* and completed successfully, then it is an error to restart it. An ad-hoc request needs to be distinguished

from previous runs by adding a unique job parameter. In either case a new *JobExecution* is created and stored to monitor this execution of the *JobInstance*.

Spring Batch Launch Environment-

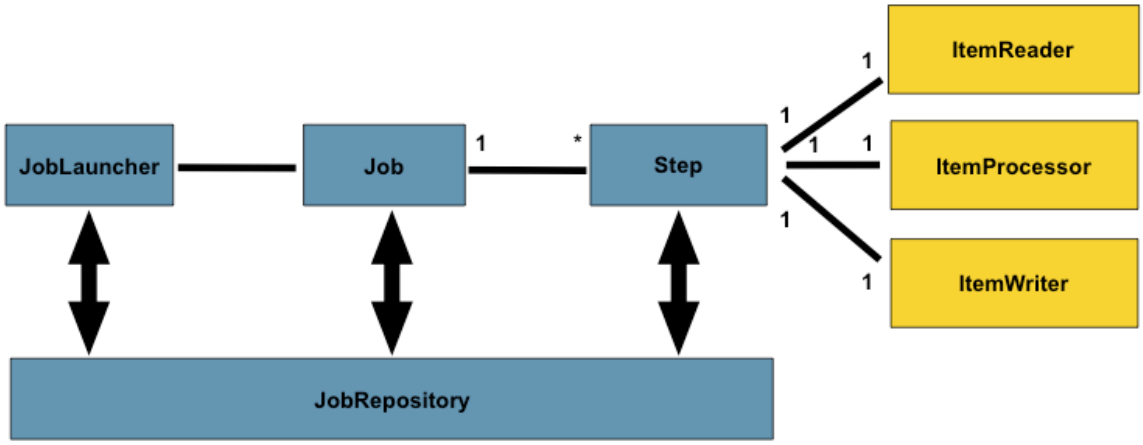


The application style is organized into four logical tiers, which include **Run, Job, Application, and Data** tiers. The primary goal for organizing an application according to the tiers is to embed what is known as "**separation of concerns**" within the system. Effective separation of concerns results in reducing the impact of change to the system.

- **Run Tier:** The Run Tier is concerned with the scheduling and launching of the application. A vendor product is typically used in this tier to allow time-based and interdependent scheduling of batch jobs as well as providing parallel processing capabilities.
- **Job Tier:** The Job Tier is responsible for the overall execution of a batch job. It sequentially executes batch steps, ensuring that all steps are in the correct state and all appropriate policies are enforced.
- **Application Tier:** The Application Tier contains components required to execute the program. It contains specific modules that address the required batch functionality and enforces policies around a module execution (e.g., commit intervals, capture of statistics, etc.)

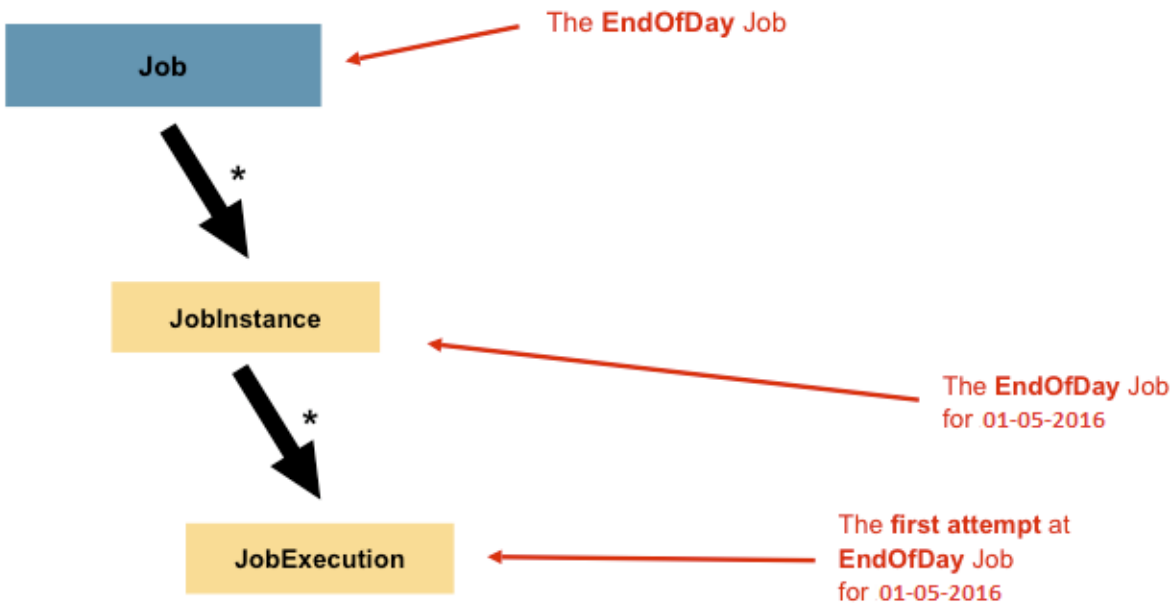
- **Data Tier:** The Data Tier provides the integration with the physical data sources that might include databases, files, or queues. Note: In some cases the Job tier can be completely missing and in other cases one job script can start several batch job instances.

Spring Batch: Core Concepts



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

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. This file 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 and allows for configuration of properties global to all steps, such as restartability. The job configuration contains:

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

A default simple implementation of the ***Job*** interface is provided by Spring Batch in the form of the ***SimpleJob*** class which creates some standard functionality on top of `Job`, however the batch namespace abstracts away the need to instantiate it directly. Instead, the <***job***> tag can be used:

```
<job id="myEmpExpireJob">
    <!-- Step bean details ommitted for clarity -->
    <step id="readEmployeeData" next="writeEmployeeData"></step>
    <step id="writeEmployeeData" next="employeeDataProcess"></step>
    <step id="employeeDataProcess"></step>
</job>
```

JobInstance

A JobInstance refers to the concept of a logical job run. Let's consider a batch job that should be run once at the end of the day, such as the 'EndOfDay' job from the diagram above. There is one 'EndOfDay' Job, but each individual run of the Job must be tracked separately. In the case of this job, there will be one logical JobInstance per day. For example, there will be a January

1st run, and a January 2nd run. If the January 1st run fails the first time and is run again the next day, it is still the January 1st run.

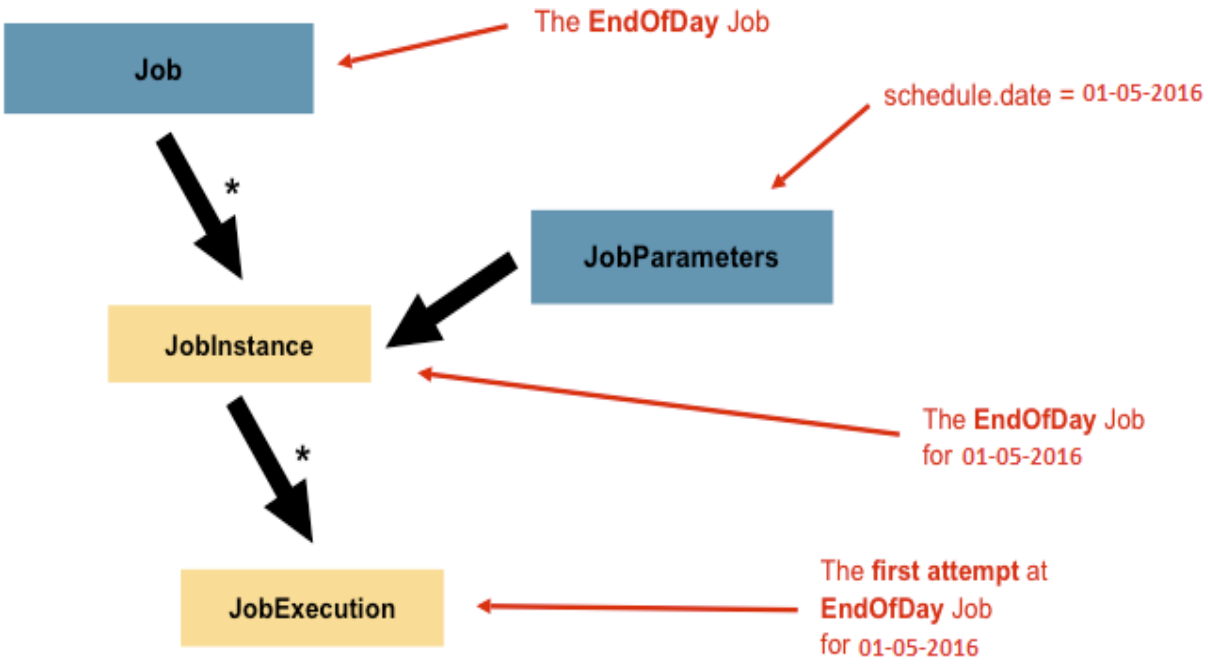
JobParameters

having discussed **JobInstance** and how it differs from Job, the natural question to ask is :-

Q :- **how is one JobInstance distinguished one from another**?

Ans :-The answer is: **JobParameters**.

JobParameters is a set of parameters used to start a batch job. They can be used for identification or even as reference data during the run:



JobExecution

A ***JobExecution*** refers to the technical concept of a single attempt to run a Job. An execution may end in failure or success, but the ***JobInstance*** corresponding to a given execution will not be considered complete unless the execution completes successfully. Using the EndOfDay Job described above as an example, consider a ***JobInstance*** for 01-05-2016 that failed the first time it was run. If it is run again with the same job parameters as the first run (01-05-2016), a new ***JobExecution*** will be created. However, there will still be only one ***JobInstance***.

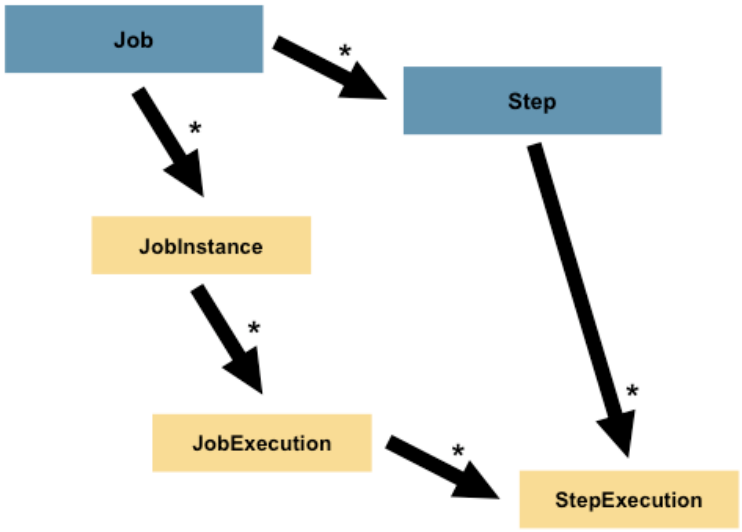
A Job defines what a job is and how it is to be executed, and JobInstance is a purely organizational object to group executions together, primarily to enable correct restart semantics. A JobExecution, however, is the primary storage mechanism for what actually happened during a run, and as such contains many more properties that must be controlled and persisted:

status	A BatchStatus object that indicates the status of the execution. While running, it's BatchStatus.STARTED, if it fails, it's BatchStatus.FAILED, and if it finishes successfully, it's BatchStatus.COMPLETED
startTime	A java.util.Date representing the current system time when the execution was started.
endTime	A java.util.Date representing the current system time when the execution finished, regardless of whether or not it was successful.
exitStatus	The ExitStatus indicating the result of the run. It is most important because it contains an exit code that will be returned to the caller.
createTime	A java.util.Date representing the current system time when the JobExecution was first persisted. The job may not have been started yet (and thus has no start time), but it will always have a createTime, which is required by the framework for managing job level ExecutionContexts.
lastUpdated	A java.util.Date representing the last time a JobExecution was persisted.
executionContext	The 'property bag' containing any user data that needs to be persisted between executions.
failureExceptions	The list of exceptions encountered during the execution of a Job. These can be useful if more than one exception is encountered during the failure of a Job.

Step-

A ***Step*** is a domain object that encapsulates an independent, sequential phase of a batch job. Therefore, every Job is composed entirely of one or more steps. A Step contains all of the information necessary to define and control the actual batch processing.

A Step can be as simple or complex as the developer desires. A simple Step might load data from a file into the database, requiring little or no code. (Depending upon the implementations used) A more complex Step may have complicated business rules that are applied as part of the processing. As with Job, a Step has an individual ***StepExecution*** that corresponds with a unique ***JobExecution***:



StepExecution-

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

Step executions are represented by objects of the StepExecution class. Each execution contains a reference to its corresponding step and JobExecution, and transaction related data such as commit and rollback count and start and end times. Additionally, each step execution will contain an ExecutionContext, which contains any data a developer needs persisted across batch runs, such as statistics or state information needed to restart. The following is a listing of the properties for StepExecution:

StepExecution Properties :-

status	A `BatchStatus` object that indicates the status of the execution. While it's running, the status is BatchStatus.STARTED, if it fails, the status is BatchStatus.FAILED, and if it finishes successfully, the status is BatchStatus.COMPLETED
startTime	A `java.util.Date` representing the current system time when the execution was started.
endTime	A `java.util.Date` representing the current system time when the execution finished, regardless of whether or not it was successful.
exitStatus	The `ExitStatus` indicating the result of the execution. It is most important because it contains an exit code that will be returned to the caller. See chapter 5 for more details.

executionContext	The 'property bag' containing any user data that needs to be persisted between executions.
readCount	The number of items that have been successfully read
writeCount	The number of items that have been successfully written
commitCount	The number transactions that have been committed for this execution
rollbackCount	The number of times the business transaction controlled by the `Step` has been rolled back.
readSkipCount	The number of times `read` has failed, resulting in a skipped item.
processSkipCount	The number of times `process` has failed, resulting in a skipped item.
filterCount	The number of items that have been 'filtered' by the `ItemProcessor`.
writeSkipCount	The number of times `write` has failed, resulting in a skipped item.

As stated above, each Job must have one or more steps in it. So the actual processing that goes on in a Job is contained in a Step. Steps can be processed in either of the following two ways.

- Chunks
- Tasklets

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:

```
<job-repository id="jobRepository"/>
```

JobLauncher-

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

```
1. public interface JobLauncher {
2. 
3.     public JobExecution run(Job job, JobParameters jobParameters)
4.                 throws JobExecutionAlreadyRunningException, JobRestartException;
5. }
```

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*** interface and its various implementations can be found in later Chapters.

Item Writer-

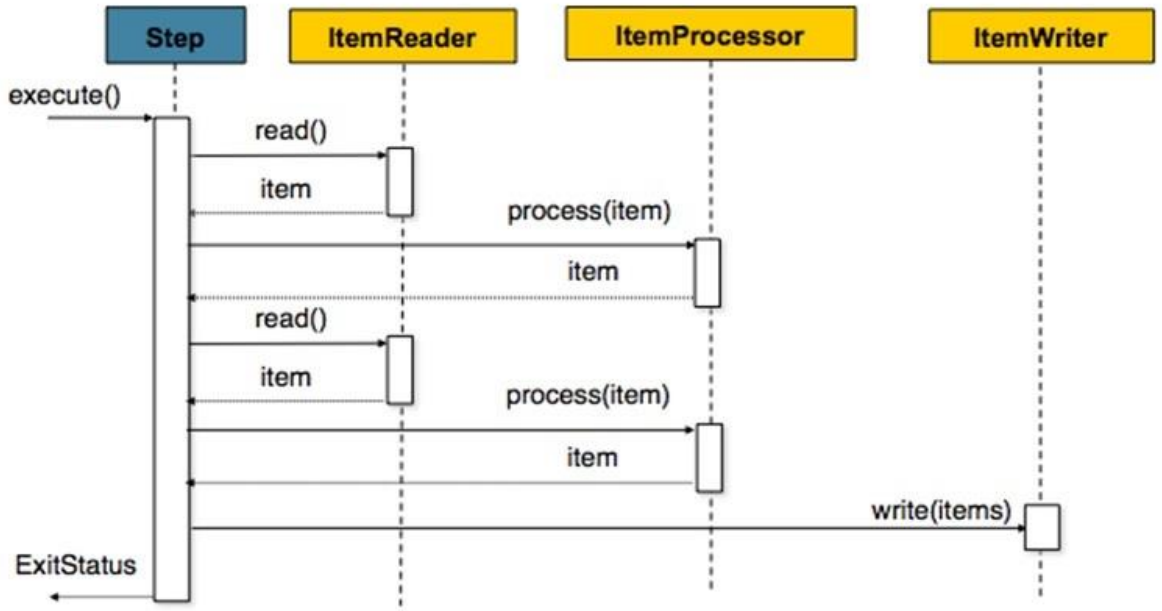
ItemWriter is an abstraction that represents the output of a Step, one batch or chunk of items at a time. Generally, an item writer has no knowledge of the input it will receive next, only the item that was passed in its current invocation. More details about the ***ItemWriter*** interface and its various implementations can be found in later Chapters.

Item Processor-

ItemProcessor is an abstraction that represents the business processing of an item. While the ***ItemReader*** reads one item, and the ***ItemWriter*** writes them, the ***ItemProcessor*** provides access to transform or apply other business processing. If, while processing the item, it is determined that the item is not valid, returning null indicates that the item should not be written out. More details about the ***ItemProcessor*** interface can be found in later Chapters.

Chunks

Chunk-oriented processing is the most common mode of Step processing. It involves Reading an input, Processing the input through the application of some business logic and aggregating it till the `commit-interval` is reached and finally writing out the `chunk` of data output to a file or database table. A bunch of Readers and Writers are floated by the framework that could be used as well as customized. The following diagram nicely summarizes the concept.



And following snippet shows how one could configure a chunk-oriented step.

```
<job id="sampleJob" job-repository="myJobRepository">
    <step id="step1">
        <tasklet transaction-manager="myTransactionManager">
            <chunk reader="itemReader" writer="itemWriter" commit-interval="10"/>
        </tasklet>
    </step>
</job>
```

Points to note on the above configuration:

- `itemReader` and `itemWriter` would need to be supplied.
- Providing an `itemProcessor` is optional
- Here a `commit-interval` of 10 implies, 10 records would be read, one-by-one and then the whole chunk of 10 records would be written off at one go.

Tasklets

`TaskletStep` processing comes to the fore when Step processing does not involve Reading or Processing and Writing, but say, just executing one stored procedure or making a remote call or just one task. The following shows how to configure a `TaskletStep`.

```
<job id="taskletJob">
    <step id="callStoredProc">
       <tasklet ref="callingProc"/>
    </step>
</job>
```

Batch Namespace-

Many of the domain concepts listed above need to be configured in a Spring ***ApplicationContext.*** While there are implementations of the interfaces above that can be used in a standard bean definition, a namespace has been provided for ease of configuration:

```
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.springframework.org/schema/batch" xsi:schemalocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
            http://www.springframework.org/schema/batch
            http://www.springframework.org/schema/batch/spring-batch-2.0.xsd">

```

ItemReader-

Although a simple concept, an ***ItemReader*** is the means for providing data from many different types of input. The most general examples include:

- Flat File- Flat File Item Readers read lines of data from a flat file that typically describe records with fields of data defined by fixed positions in the file or delimited by some special character (e.g. Comma).
- XML - XML ***ItemReaders*** process XML independently of technologies used for parsing, mapping and validating objects. Input data allows for the validation of an XML file against an XSD schema.
- Database - A database resource is accessed to return resultsets which can be mapped to objects for processing. The default SQL ***ItemReaders*** invoke a ***RowMapper*** to return

objects, keep track of the current row if restart is required, store basic statistics, and provide some transaction enhancements that will be explained later.

There are many more possibilities, but we'll focus on the basic ones for this chapter.
A complete list of all available ***ItemReaders*** are

1. AmqpItemReader
2. AggregateItemReader
3. FlatFileItemReader
4. HibernateCursorItemReader
5. HibernatePagingItemReader
6. IbatisPagingItemReader
7. ItemReaderAdapter
8. JdbcCursorItemReader
9. JdbcPagingItemReader
10. JmsItemReader
11. JpaPagingItemReader
12. ListItemReader
13. MongoItemReader
14. Neo4jItemReader
15. RepositoryItemReader
16. StoredProcedureItemReader
17. StaxEventItemReader

We can see that Spring Batch already provides readers for many of the formatting standards and database industry providers. It is recommended to use the abstractions provided by Spring Batch in your applications rather than creating your own ones.

ItemReader is a basic interface for generic input operations:

```
1. public interface ItemReader<T> {
2.
3.     T read() throws Exception, UnexpectedInputException, ParseException;
4.
5. }
```

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. However, if the underlying resource is transactional (such as a JMS queue) then calling read may return the same logical item on subsequent calls in a rollback scenario. It is also worth noting that a

lack of items to process by an ***ItemReader*** will not cause an exception to be thrown. For example, a database ***ItemReader*** that is configured with a query that returns 0 results will simply return null on the first invocation of read.

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.

A complete list of all available ***ItemWriter*** are :-

1. AbstractItemStreamItemWriter
2. AmqpItemWriter
3. CompositeItemWriter
4. FlatFileItemWriter
5. GemfireItemWriter
6. HibernateItemWriter
7. IbatisBatchItemWriter
8. ItemWriterAdapter
9. JdbcBatchItemWriter
10. JmsItemWriter
11. JpaItemWriter
12. MimeMessageItemWriter
13. MongoItemWriter
14. Neo4jItemWriter
15. StaxEventItemWriter
16. RepositoryItemWriter

We can see that Spring Batch already provides Writers for many of the formatting standards and database industry providers. It is recommended to use the abstractions provided by Spring Batch in your applications rather than creating your own ones.

As with ***ItemReader***, ***ItemWriter*** is a fairly generic interface:

```
1.  public interface ItemWriter<T> {
2.
3.      void write(List<? extends T> items) throws Exception;
4.
5.  }
```

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***. For example:

```
1.      public class CompositeItemWriter<T> implements ItemWriter<T>  {
2.
3.          ItemWriter<T> itemWriter;
4.
5.          public CompositeItemWriter(ItemWriter<T>  itemWriter) {
6.              this.itemWriter = itemWriter;
7.          }
8.
9.          public void write(List<? extends T>  items) throws Exception
{
10.             //Add business logic here
11.            itemWriter.write(item);
12.         }
13.
14.         public void setDelegate(ItemWriter<T>  itemWriter){
15.             this.itemWriter = itemWriter;
```



```
16.             }
17.         }
```

The class above contains another ***ItemWriter*** to which it ***delegates*** after having provided some business logic. This pattern could easily be used for an ***ItemReader*** as well, perhaps to obtain more reference data based upon the input that was provided by the main ***ItemReader***. It is also useful if you need to control the call to write yourself. However, if you only want to 'transform' the item passed in for writing before it is actually written, there isn't much need to call write yourself: you just want to modify the item. For this scenario, Spring Batch provides the ***ItemProcessor*** interface:

```
1.          public interface ItemProcessor<I, O> {
2.
3.              O process(I item) throws Exception;
4.          }
```

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:

```
1.        public class Anurag {}
2.
3.    public class JavaDeveloper {
4.        public JavaDeveloper(Anurag anu) {}
5.    }
6.    public class AnuragProcessor implements ItemProcessor<Anurag, JavaDeveloper>{
7.            public JavaDeveloper process(Anurag anu) throws Exception {
8.       //Perform simple transformation, convert Anurag to a JavaDeveloper
9.            return new JavaDeveloper(anu);
10.       }
11.   }
12.
13.   public class JavaDeveloperWriter implements ItemWriter<JavaDeveloper>{
14.       public void write(List<? extends JavaDeveloper> devlopers) throws Exception {
15.           //write devlopers
16.       }
17.   }
```

In the very simple example above, there is a class Anurag, a class JavaDeveloper, and a class **AnuragProcessor** that implements to the **ItemProcessor** interface. The transformation is

simple, but any type of transformation could be done here. The **JavaDeveloperWriter** will be used to write out JavaDeveloper objects, throwing an exception if any other type is provided. Similarly, the **AnuragProcessor** will throw an exception if anything but an Anurag is provided. The **AnuragProcessor** can then be injected into a Step:

```
1.       <job id="ioSampleJob">
2.        <step name="step1">
3.            <tasklet>
4.                <chunk reader="anuragReader" processor="anuragProcessor"  writer="javaDeveloperWriter"
5.                       commit-interval="2"/>
6.               </tasklet>
7.           </step>
8.    </job>
```

Configuring and Running a Job in Spring Batch

In this chapter we will explain the various configuration options and run time concerns of a Job. While the Job object may seem like a simple container for steps, there are many configuration options of which a developers must be aware. Furthermore, there are many considerations for how a Job will be run and how its meta-data will be stored during that run.

Configuring a Job-

There are multiple implementations of the Job interface, however, the namespace abstracts away the differences in configuration. It has only three required dependencies:
a **name**, **JobRepository,** and a list of **Steps**.

```
1.  <job id="myEmpExpireJob">
2.      <step id="readEmployeeData" next="writeEmployeeData" parent="s1"></step>
3.      <step id="writeEmployeeData" next="employeeDataProcess" parent="s2"></step>
4.      <step id="employeeDataProcess" parent="s3"></step>
5.  </job>
```

The namespace defaults to referencing a repository with an id of '**jobRepository**', which is a sensible default. However, this can be overridden explicitly:


```
<job id="myEmpExpireJob" job-repository="specialRepository">
    <step id="readEmployeeData" next="writeEmployeeData" parent="s1"></step>
    <step id="writeEmployeeData" next="employeeDataProcess" parent="s2"></step>
    <step id="employeeDataProcess" parent="s3"></step>
</job>
```

1. Restartability-

One key issue when executing a batch job concerns the behavior of a Job when it is restarted. The launching of a Job is considered to be a 'restart' if a **JobExecution** already exists for the particular **JobInstance**. 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 **JobInstance** is created in this scenario. However, Spring Batch does provide some help. If a Job should never be restarted, but should always be run as part of a new **JobInstance**, then the restartable property may be set to 'false':

```
<job id="myEmpExpireJob" restartable="false">
    ...
</job>
```

Another way...

```
Job job = new SimpleJob();
job.setRestartable(false);

JobParameters jobParameters = new JobParameters();

JobExecution firstExecution = jobRepository.createJobExecution(job, jobParameters);
jobRepository.saveOrUpdate(firstExecution);

try {
    jobRepository.createJobExecution(job, jobParameters);
    fail();
}
catch (JobRestartException e) {
    // expected
}
```

2. Intercepting Job Execution-

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 **SimpleJob** allows for this by calling a **JobListener** at the appropriate time:


```
1. public interface JobExecutionListener {
2. 
3.     void beforeJob(JobExecution jobExecution);
4. 
5.     void afterJob(JobExecution jobExecution);
6. 
7. }
```

JobListeners can be added to a **SimpleJob** via the listener's element on the job:

```
1. <job id="myEmpExpireJob">
2.     <step id="readEmployeeData" next="writeEmployeeData" parent="s1"></step>
3.     <step id="writeEmployeeData" next="employeeDataProcess" parent="s2"></step>
4.     <step id="employeeDataProcess" parent="s3"></step>
5.     <listeners>
6.         <listener ref="sampleListener"></listener>
7.     </listeners>
8. </job>
```

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 the **JobExecution**:

```
1. public void afterJob(JobExecution jobExecution){
2.     if( jobExecution.getStatus() == BatchStatus.COMPLETED ){
3.         //job success
4.     }
5.     else if(jobExecution.getStatus() == BatchStatus.FAILED){
6.         //job failure
7.     }
8. }
```

The annotations corresponding to this interface are:

- **@BeforeJob**
- **@AfterJob**

3. 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.

In the following example, "**baseJob**" is an abstract Job definition that defines only a list of listeners. The Job "**job1**" is a concrete definition that inherits the list of listeners from "**baseJob**" and merges it with its own list of listeners to produce a Job with two listeners and one Step, "**step1**"-

```
<job abstract="true" id="baseJob">
    <listeners>
        <listener ref="listenerOne"/>
    <listeners>
</listeners></listeners></job>

<job id="job1" parent="baseJob">
    <step id="step1" parent="standaloneStep"/>

    <listeners merge="true">
        <listener ref="listenerTwo"/>
    <listeners>
</listeners></listeners></job>
```

4. JobParametersValidator-

A job declared in the XML namespace or using any subclass of ***AbstractJob*** can optionally declare a validator for the job parameters at runtime. This is useful when for instance you need to assert that a job is started with all its mandatory parameters. There is a ***DefaultJobParametersValidator*** that can be used to constrain combinations of simple mandatory and optional parameters, and for more complex constraints you can implement the interface yourself. The configuration of a validator is supported through the XML namespace through a child element of the job, e.g.:

```
<job id="job1" parent="baseJob3">
    <step id="step1" parent="standaloneStep"/>
    <validator ref="paremetersValidator"/>
</job>
```

Configuring a JobRepository-

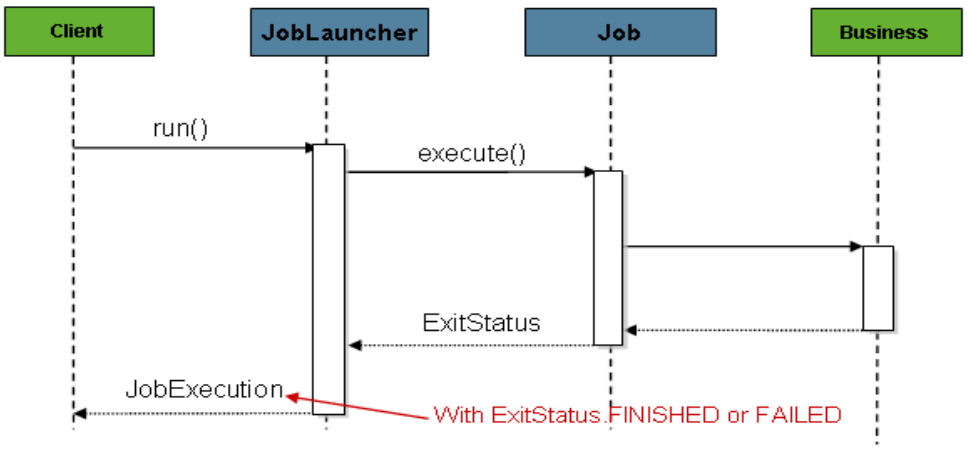
As described in earlier, the ***JobRepository*** is used for basic CRUD operations of the various persisted domain objects within Spring Batch, such as ***JobExecution*** and ***StepExecution***. It is required by many of the major framework features, such as the***JobLauncher***, Job, and Step. The batch namespace abstracts away many of the implementation details of the ***JobRepository*** implementations and their collaborators. However, there are still a few configuration options available:


```
<job-repository id="jobRepository"
    data-source="dataSource"
    transaction-manager="transactionManager"
    isolation-level-for-create="SERIALIZABLE"
    table-prefix="BATCH_"
 max-varchar-length="1000"
/>
```

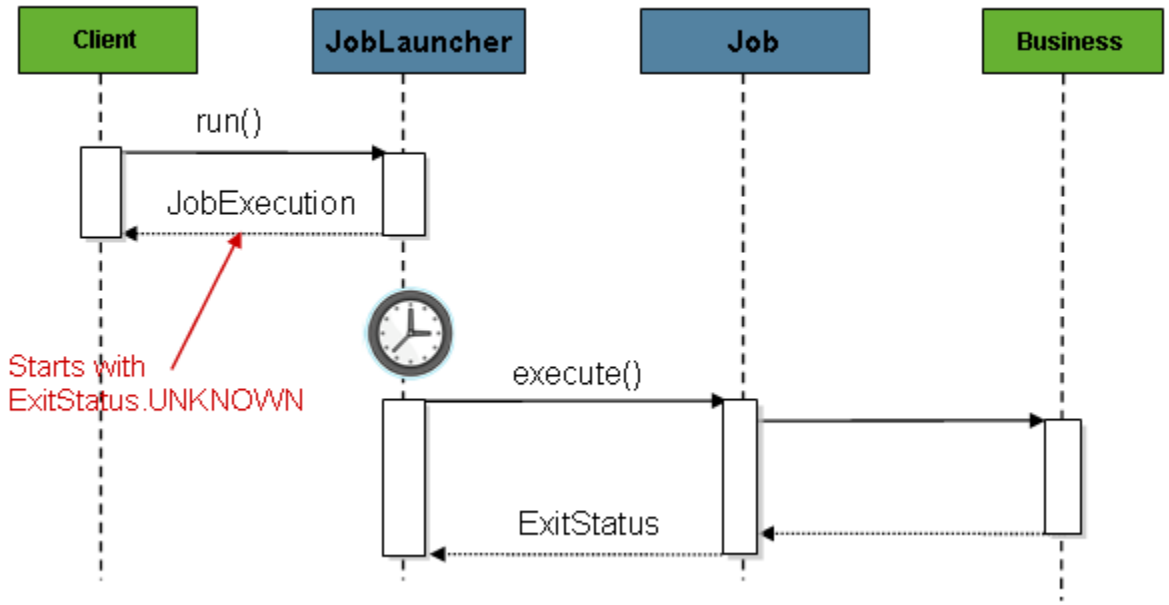
Configuring a JobLauncher-

The most basic implementation of the ***JobLauncher*** interface is the ***SimpleJobLauncher***. Its only required dependency is a ***JobRepository***, in order to obtain an execution:

```
<bean class="org.springframework.batch.core.launch.support.SimpleJobLauncher"
id="jobLauncher">
    <property name="jobRepository" ref="jobRepository" />
</bean>
```



The sequence is straightforward and works well when launched from a scheduler. However, issues arise when trying to launch from an HTTP request. In this scenario, the launching needs to be done asynchronously so that the ***SimpleJobLauncher*** returns immediately to its caller. This is because it is not good practice to keep an HTTP request open for the amount of time needed by long running processes such as batch. An example sequence is below:



The ***SimpleJobLauncher*** can easily be configured to allow for this scenario by configuring a ***TaskExecutor***

```
<bean class="org.springframework.batch.core.launch.support.SimpleJobLauncher"
id="jobLauncher">
    <property name="jobRepository" ref="jobRepository" />
    <property name="taskExecutor">
        <bean class="org.springframework.core.task.SimpleAsyncTaskExecutor" />
    </property>
</bean>
```

Any implementation of the spring ***TaskExecutor*** interface can be used to control how jobs are asynchronously executed.

Running a Job-

At a minimum, launching a batch job requires two things: the Job to be launched and a ***JobLauncher***. Both can be contained within the same context or different contexts. For example, if launching a job from the command line, a new **JVM** will be instantiated for each Job, and thus every job will have its own ***JobLauncher***. However, if running from within a web container within the scope of an ***HttpRequest***, there will usually be one ***JobLauncher***, configured for asynchronous job launching, that multiple requests will invoke to launch their jobs.

1-Running Jobs from the Command Line

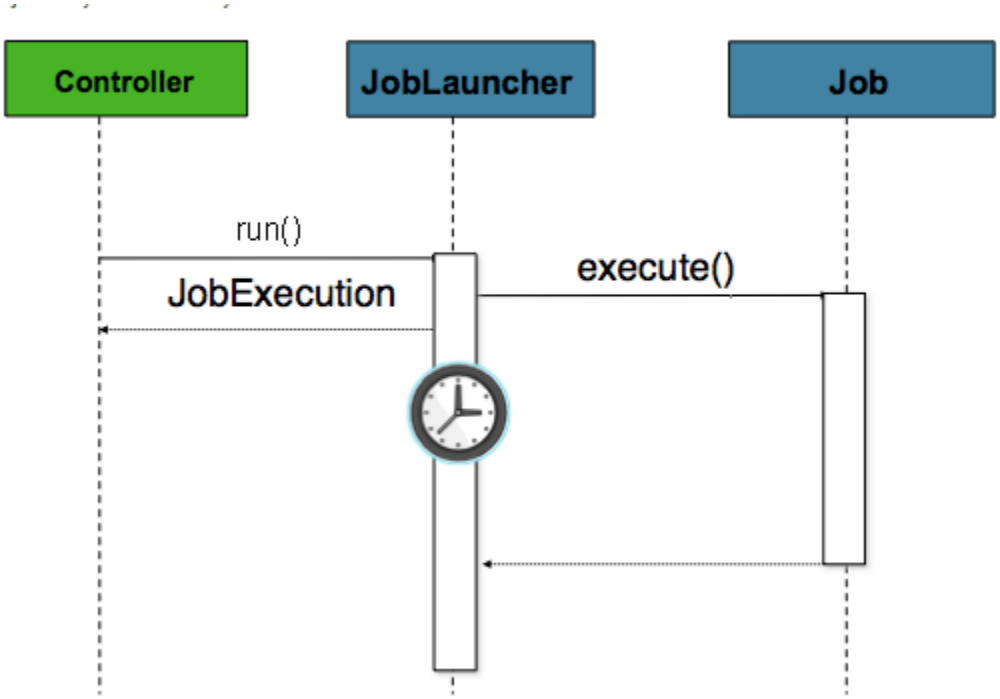
The CommandLineJobRunner- Because the script launching the job must kick off a Java Virtual Machine, there needs to be a class with a main method to act as the primary entry point. Spring Batch provides an implementation that serves just this purpose: ***CommandLineJobRunner***. It's important to note that this is just one way to bootstrap your application, but there are many ways to launch a Java process, and this class should in no way be viewed as definitive.
The ***CommandLineJobRunner*** performs four tasks:

- Load the appropriate ***ApplicationContext***
- Parse command line arguments into ***JobParameters***
- Locate the appropriate job based on arguments
- Use the ***JobLauncher*** provided in the application context to launch the job.

```
bash$ java CommandLineJobRunner endOfDayJob.xml endOfDay
schedule.date(date)=2016/01/05
```

2-Running Jobs from within a Web Container

Historically, offline processing such as batch jobs have been launched from the command-line, as described above. However, there are many cases where launching from an HttpRequest is a better option. Many such use cases include reporting, ad-hoc job running, and web application support. Because a batch job by definition is long running, the most important concern is ensuring to launch the job asynchronously:

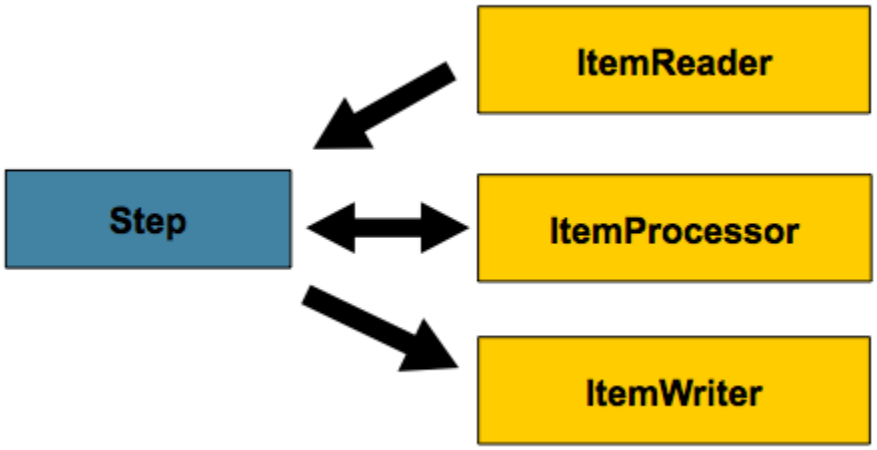


The controller in this case is a Spring MVC controller.

```
1.  @Controller
2.  public class JobLauncherController {
3.
4.      @Autowired
5.      JobLauncher jobLauncher;
6.
7.      @Autowired
8.      Job job;
9.
10.     @RequestMapping("/jobLauncher.html")
11.     public void handle() throws Exception{
12.         jobLauncher.run(job, new JobParameters());
13.     }
14. }
```

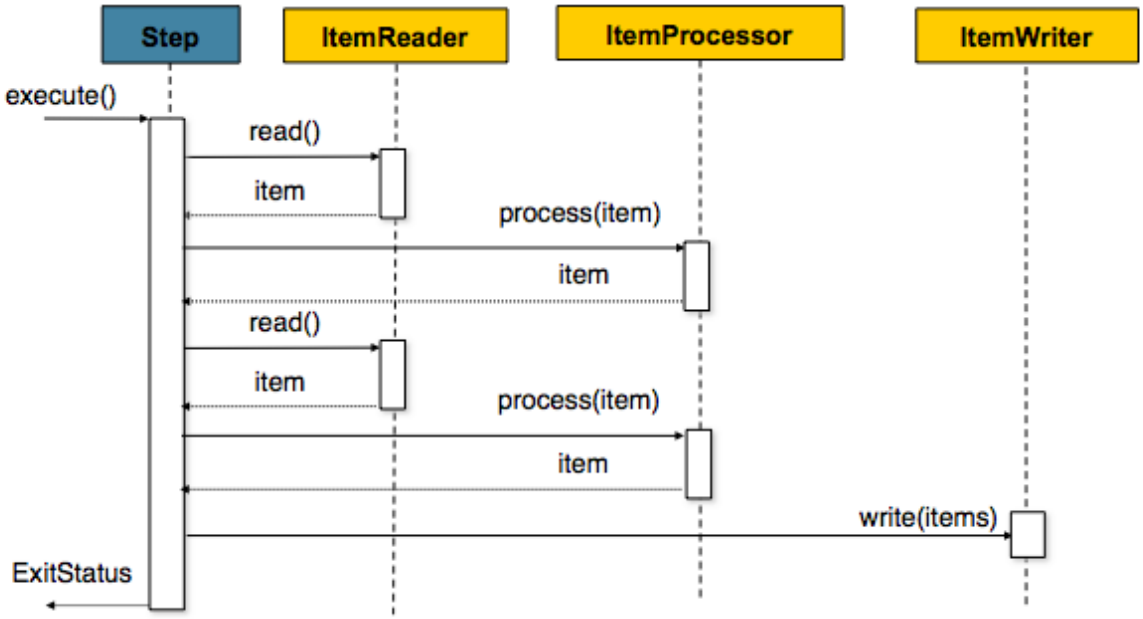
Configuring a Step in Spring Batch

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. This is a necessarily vague description because the contents of any given **Step** are at the discretion of the developer writing a Job. A Step can be as simple or complex as the developer desires. A simpleStep 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.



Chunk-Oriented Processing-

Spring Batch uses a 'Chunk Oriented' processing style within its most common implementation. Chunk oriented processing refers to reading the data one at a time, and creating 'chunks' that will be written out, within a transaction boundary. One item is read in from an ***ItemReader***, handed to an ***ItemProcessor***, and aggregated. Once the number of items read equals the commit interval, the entire chunk is written out via the ***ItemWriter***, and then the transaction is committed.



Below is a code representation of the same concepts shown above:-

```
List items = new Arraylist();
for(int i = 0; i < commitInterval; i++){
    Object item = itemReader.read()
    Object processedItem = itemProcessor.process(item);
    items.add(processedItem);
}
itemWriter.write(items);
```

1. Configuring a Step-

```
<job id="sampleJob" job-repository="jobRepository">
    <step id="step1">
        <tasklet transaction-manager="transactionManager">
            <chunk reader="itemReader" writer="itemWriter" commit-interval="10"/>
        </tasklet>
    </step>
</job>
```

The configuration above represents the only required dependencies to create an item-oriented step:

- reader - The ItemReader that provides items for processing.
- writer - The ItemWriter that processes the items provided by the ItemReader.
- transaction-manager - Spring's Platform TransactionManager that will be used to begin and commit transactions during processing.
- 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/>.
- commit-interval - The number of items that will be processed before the transaction is committed.

It should be noted that, job-repository defaults to "jobRepository" and transaction-manager defaults to "transactionManger". Furthermore, the ItemProcessor is optional, not required, since the item could be directly passed from the reader to the writer.

2. Inheriting from a Parent Step-

If a group of Steps share similar configurations, then it may be helpful to define a

"parent" Step from which the concrete Steps may inherit properties. Similar to class inheritance in Java, the "child" Step will combine its elements and attributes with the parent's. The child will also override any of the parent's Steps.
In the following 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***":

```
<step id="parentStep">
    <tasklet allow-start-if-complete="true">
       <chunk reader="itemReader" writer="itemWriter" commit-interval="10"/>
    </tasklet>
<step id="concreteStep1" parent="parentStep">
<tasklet start-limit="5">
<chunk processor="itemProcessor" commit-interval="5"/>
</tasklet>
</step>
```

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***.


```
<step abstract="true" id="abstractParentStep">
    <tasklet>
       <chunk commit-interval="10"/>
    </tasklet>
</step>

<step id="concreteStep2" parent="abstractParentStep">
    <tasklet>
        <chunk reader="itemReader" writer="itemWriter"/>
    </tasklet>
</step>
```

3. The Commit Interval-

As mentioned above, a step reads in and writes out items, periodically committing using the supplied ***PlatformTransactionManager***. With a commit-interval of 1, it will commit after writing each individual item. This is less than ideal in many situations, since beginning and committing a transaction is expensive. Ideally, it is preferable to process as many items as possible in each transaction, which is completely dependent upon the type of data being processed and the resources with which the step is interacting. For this reason, the number of items that are processed within a commit can be configured.

```
1. <job id="sampleJob">
2.     <step id="step1">
3.         <tasklet>
4.             <chunk reader="itemReader" writer="itemWriter" commit-interval="10"/>
5.         </tasklet>
6.     </step>
7. </job>
```

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

4. Configuring a Step for Restart-

- **Setting a StartLimit -** There are many scenarios where you may want to control the number of times a Step may be started. For example, a particular Step might need

to be configured so that it only runs once because it invalidates some resource that must be fixed manually before it can be run again.

```
<step id="step1">
    <tasklet start-limit="1">
       <chunk reader="itemReader" writer="itemWriter" commit-interval="10"/>
    </tasklet>
</step>
```

The simple step above can be run only once. Attempting to run it again will cause an exception to be thrown. It should be noted that the default value for the start-limit is Integer.MAX_VALUE.

- **Restarting a completed step-**

```
<step id="step1">
    <tasklet allow-start-if-complete="true">
        <chunk reader="itemReader" writer="itemWriter" commit-interval="10"/>
    </tasklet>
</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.

5. Configuring Skip Logic-

There are many scenarios where errors encountered while processing should not result in Step failure, but should be skipped instead. This is usually a decision that must be made by someone who understands the data itself and what meaning it has. Financial data, for example, may not be skippable because it results in money being transferred, which needs to be completely accurate. Loading a list of vendors, on the other hand, might allow for skips. If a vendor is not loaded because it was formatted incorrectly or was missing necessary information, then there probably won't be issues. Usually these bad records are logged as well, which will be covered later when discussing listeners.


```
<step id="step1">
   <tasklet>
      <chunk commit-interval="10" reader="flatFileItemReader" skip-limit="10"
writer="itemWriter">
         <skippable-exception-classes>
           <include class="org.springframework.batch.item.file.FlatFileParseException"/>
         </skippable-exception-classes>
      </chunk>
   </tasklet>
</step>
```

In this example, a ***FlatFileItemReader*** is used, and if at any point a ***FlatFileParseException*** is thrown, it will be skipped and counted against the total skip limit of 10. Separate counts are made of skips on read, process and write inside the step execution, and the limit applies across all. Once the skip limit is reached, the next exception found will cause the step to fail.

```
<step id="step1">
    <tasklet>
        <chunk commit-interval="10" reader="flatFileItemReader" skip-limit="10"
writer="itemWriter">
            <skippable-exception-classes>
                <include class="java.lang.Exception"/>
                <exclude class="java.io.FileNotFoundException"/>
            </skippable-exception-classes>
        </chunk>
    </tasklet>
</step>
```

By 'including' java.lang.Exception as a skippable exception class, the configuration indicates that all Exceptions are skippable. However, by 'excluding' java.io.FileNotFoundException, the configuration refines the list of skippable exception classes to be all Exceptions except FileNotFoundException. Any excluded exception classes will be fatal if encountered (i.e. not skipped).

6. Configuring Retry Logic-

In most cases you want an exception to cause either a skip or Step failure. However, not all exceptions are deterministic. If a ***FlatFileParseException*** is encountered while reading, it will always be thrown for that record; resetting the ***ItemReader*** will not help. However, for other exceptions, such as a ***DeadlockLoserDataAccessException***, which indicates that the current process has attempted to update a record that another process holds a lock on, waiting and trying again might result in success. In this case, retry should be configured:


```
<step id="step1">
   <tasklet>
      <chunk commit-interval="2" reader="itemReader" retry-limit="3" writer="itemWriter">
         <retryable-exception-classes>
            <include class="org.springframework.dao.DeadlockLoserDataAccessException"/>
         </retryable-exception-classes>
      </chunk>
   </tasklet>
</step>
```

7. 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.

```
<step id="step1">
   <tasklet>
      <chunk commit-interval="2" reader="itemReader" writer="itemWriter">
      <no-rollback-exception-classes>
         <include class="org.springframework.batch.item.validator.ValidationException"/>
      </no-rollback-exception-classes>
   </chunk></tasklet>
</step>
```

8. Transaction Attributes-

```
    <tasklet>
        <chunk reader="itemReader" writer="itemWriter" commit-interval="2"/>
        <transaction-attributes isolation="DEFAULT"
                                propagation="REQUIRED"
                                timeout="30"/>
    </tasklet>
```

9. Registering ItemStreams with the Step -

```
<step id="step1">
    <tasklet>
        <chunk commit-interval="2" reader="itemReader" writer="compositeWriter">
            <streams>
                <stream ref="fileItemWriter1"/>
                <stream ref="fileItemWriter2"/>
            </streams>
        </chunk>
    </tasklet>
</step>

<beans:bean class="org.springframework.batch.item.support.CompositeItemWriter"
id="compositeWriter">
    <beans:property name="delegates">
        <beans:list>
            <beans:ref bean="fileItemWriter1" />
            <beans:ref bean="fileItemWriter2" />
        </beans:list>
    </beans:property>
</beans:bean>
```

10. Intercepting Step Execution-

```
<step id="step1">
    <tasklet>
          <chunk reader="reader" writer="writer" commit-interval="10"/>
        <listeners>
            <listener ref="chunkListener"/>
        </listeners>
    </tasklet>
</step>
```

Listeners :-

StepExecutionListener

```
1. public interface StepExecutionListener extends StepListener {
2.
3.     void beforeStep(StepExecution stepExecution);
4.
5.     ExitStatus afterStep(StepExecution stepExecution);
6.
7. }
```

ChunkListener

```
1. public interface ChunkListener extends StepListener {
2.
3.     void beforeChunk();
4.
5.     void afterChunk();
6.
7. }
```

ItemReadListener

```
1.  public interface ItemReadListener<T> extends StepListener {
2.
3.      void beforeRead();
4.
5.      void afterRead(T item);
6.
7.      void onReadError(Exception ex);
8.
9.  }
```

ItemProcessListener

```
1.  public interface ItemProcessListener<T, S> extends StepListener {
2.
3.      void beforeProcess(T item);
4.
5.      void afterProcess(T item, S result);
6.
7.      void onProcessError(T item, Exception e);
8.
9.  }
```

ItemWriteListener

```
1.  public interface ItemWriteListener<S> extends StepListener {
2.
3.      void beforeWrite(List<? extends S> items);
4.
5.      void afterWrite(List<? extends S> items);
6.
7.      void onWriteError(Exception exception, List<? extends S> items);
8.
9.  }
```

SkipListener

```
1.  public interface SkipListener<T,S> extends StepListener {
2.
3.      void onSkipInRead(Throwable t);
4.
5.      void onSkipInProcess(T item, Throwable t);
6.
7.      void onSkipInWrite(S item, Throwable t);
8.
9.  }
```

TaskletStep

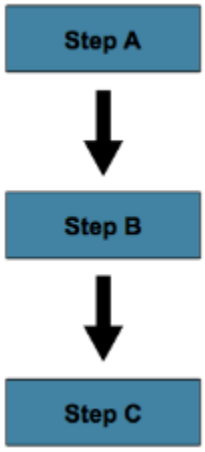
The ***Tasklet*** is a simple interface that has one method, execute, which will be a called repeatedly by the ***TaskletStep*** until it either returns ***RepeatStatus***.FINISHED or throws an exception to signal a failure. Each call to the ***Tasklet*** is wrapped in a transaction.
Tasklet implementers 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/>:***

```
<step id="step1">
    <tasklet ref="myTasklet"/>
</step>
```

Controlling Step Flow-

With the ability to group steps together within an owning job comes the need to be able to control how the job 'flows' from one step to another. The failure of a Step doesn't necessarily mean that the Job should fail. Furthermore, there may be more than one type of 'success' which determines which Step should be executed next. Depending upon how a group of Steps is configured, certain steps may not even be processed at all.

1. Sequential Flow- The simplest flow scenario is a job where all of the steps execute sequentially:



This can be achieved using the 'next' attribute of the step element:

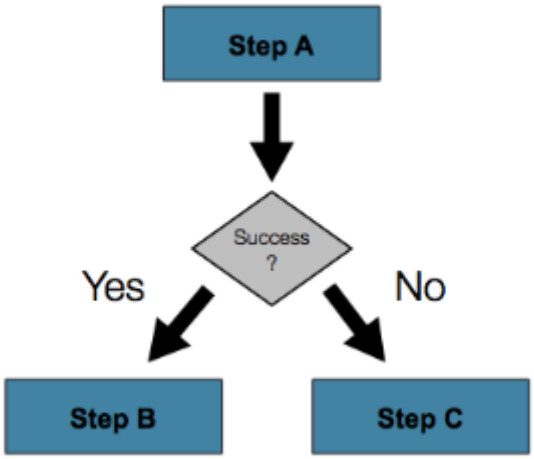
```
<job id="job">
    <step id="stepA" parent="s1" next="stepB" />
    <step id="stepB" parent="s2" next="stepC"/>
    <step id="stepC" parent="s3" />
</job>
```

2. Conditional Flow-

In the example above, there are only two possibilities:

1. The Step is successful and the next Step should be executed.
2. The Step failed and thus the Job should fail.

In many cases, this may be sufficient. However, what about a scenario in which the failure of a Step should trigger a different Step, rather than causing failure?



In order to handle more complex scenarios, the Spring Batch namespace allows transition elements to be defined within the step element. One such transition is the "next" element. Like the "next" attribute, the "next" element will tell the Job which Step to execute next. However, unlike the attribute, any number of "next" elements are allowed on a given Step, and there is no default behavior the case of failure. This means that if transition elements are used, then all of the behavior for the Step's transitions must be defined explicitly. Note also that a single step cannot have both a "next" attribute and a transition element.
The next element specifies a pattern to match and the step to execute next:


```
<job id="job">
    <step id="stepA" parent="s1">
        <next on="*" to="stepB" />
        <next on="FAILED" to="stepC" />
    </step>
    <step id="stepB" parent="s2" next="stepC" />
    <step id="stepC" parent="s3" />
</job>
```

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. The framework will automatically order transitions from most specific to least specific. This means that even if the elements were swapped for "stepA" in the example above, an ExitStatus of "FAILED" would still go to "stepC".

**********Under Progress **********

