Spring Batch

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.

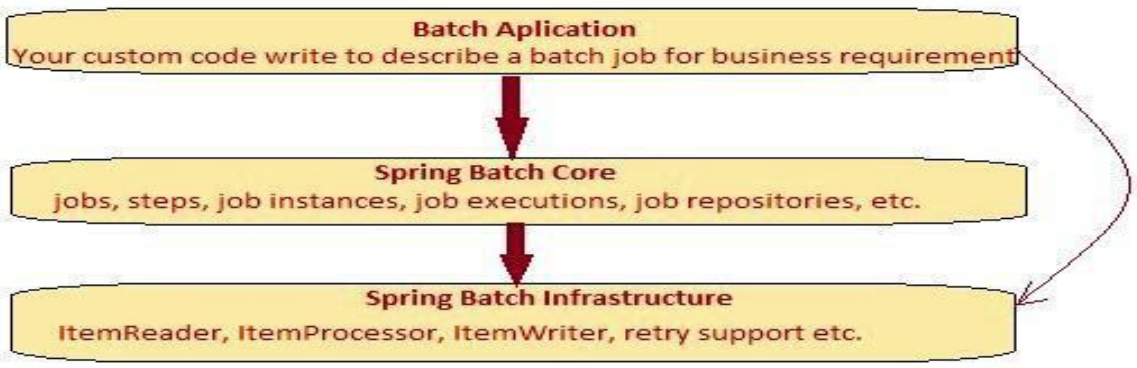
6. Normally a batch application can be divided in three main parts: □ Reading the data (from a database, file system, etc.)
□ Processing the data (filtering, grouping, calculating, validating…) □ Writing the data (to a database, reporting, distributing…)

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.

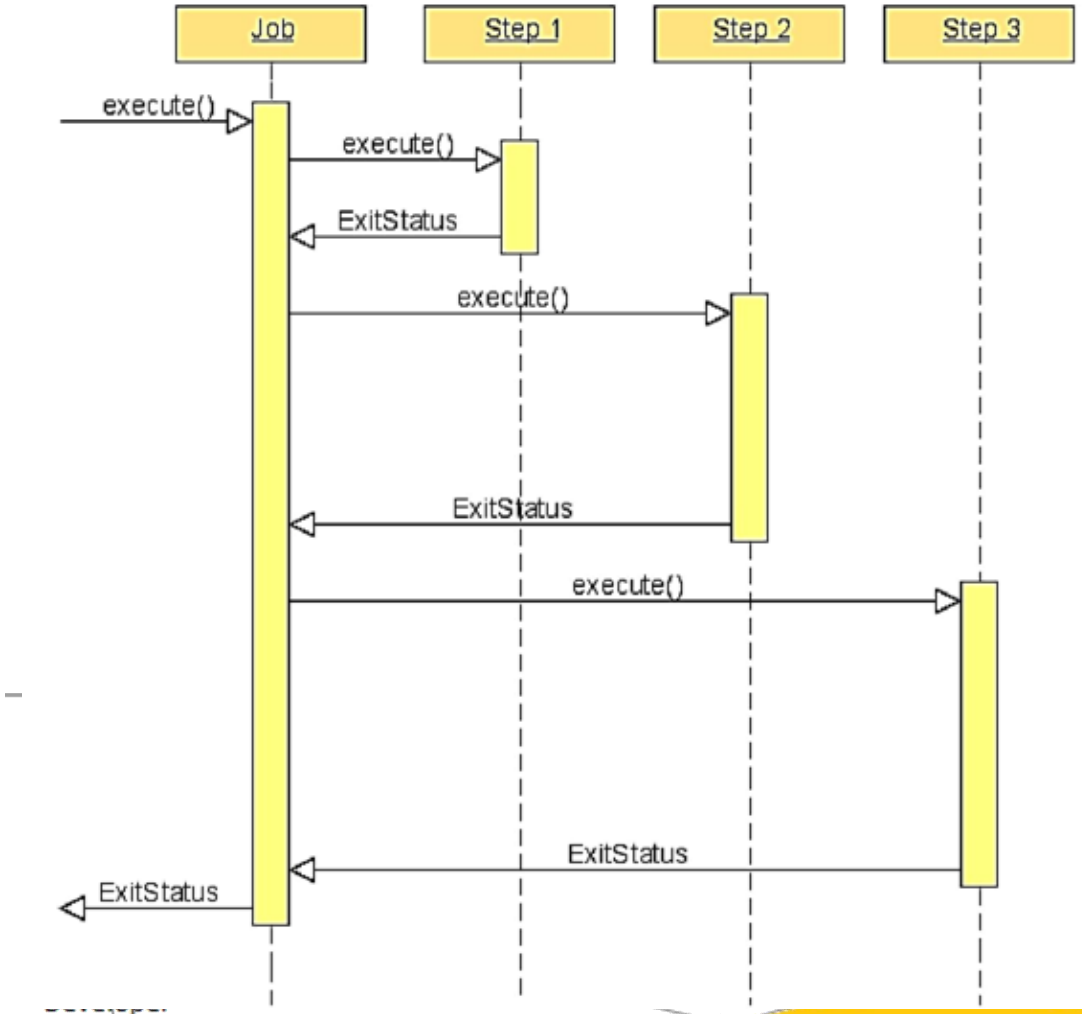
8. It also contains classes and interfaces for the main data formats, industry standards and providers like XML, CSV, SQL, Mongo DB, etc.

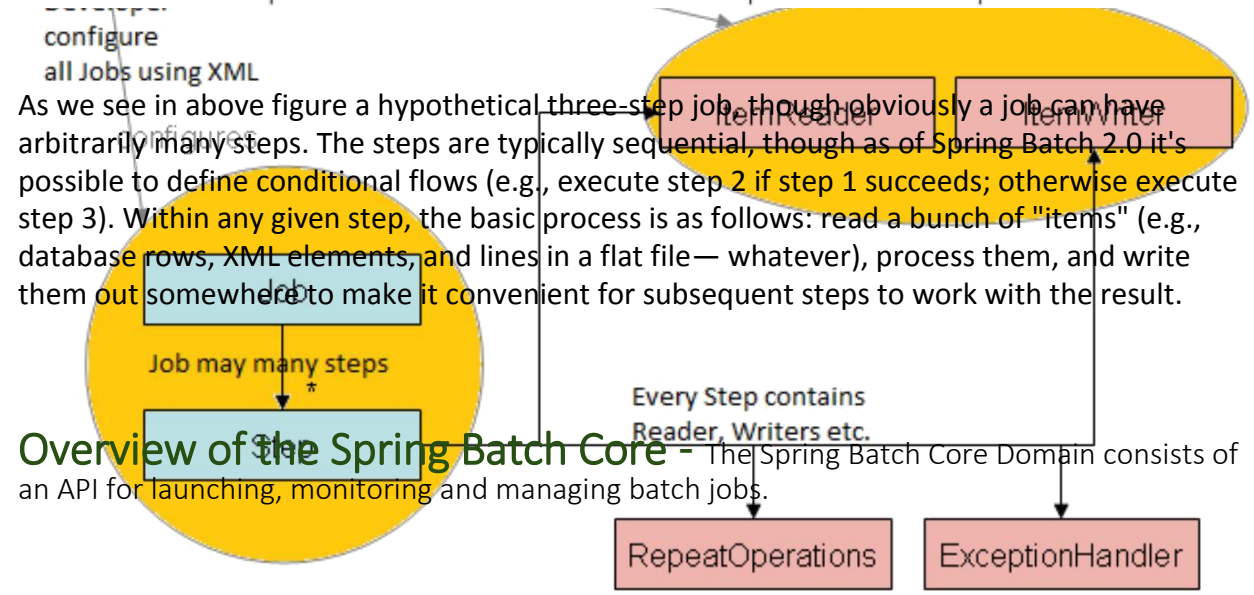
9. Examples :- Month-end calculations notices or correspondence , periodic application of complex business rules processed repetitively across very large data sets (e.g. insurance beneficiries
determination or rate adjustments)

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.

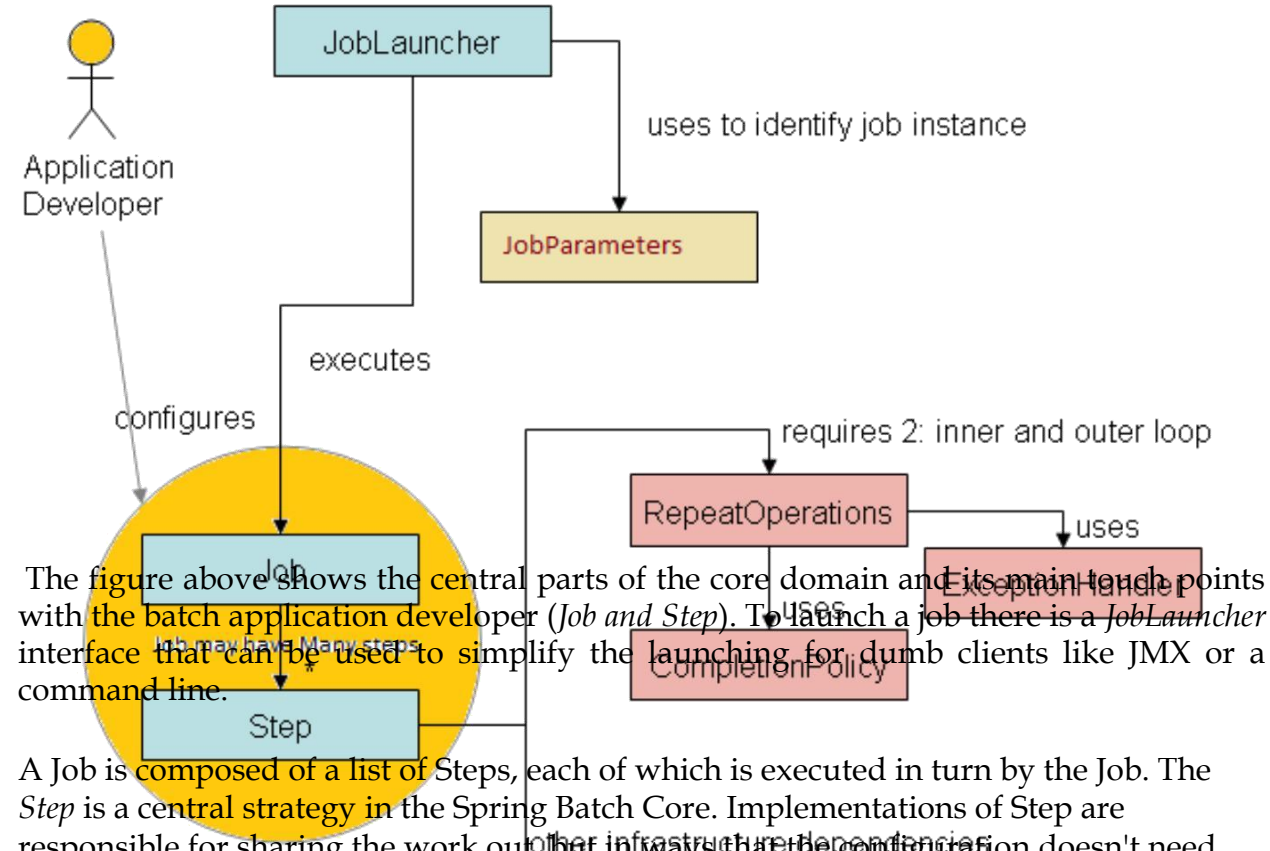




As we see in above figure a hypothetical three-step job, though obviously a job can have arbitrarily many steps. The steps are typically sequential, though as of Spring Batch 2.0 it's possible to define conditional flows (e.g., execute step 2 if step 1 succeeds; otherwise execute step 3). Within any 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.

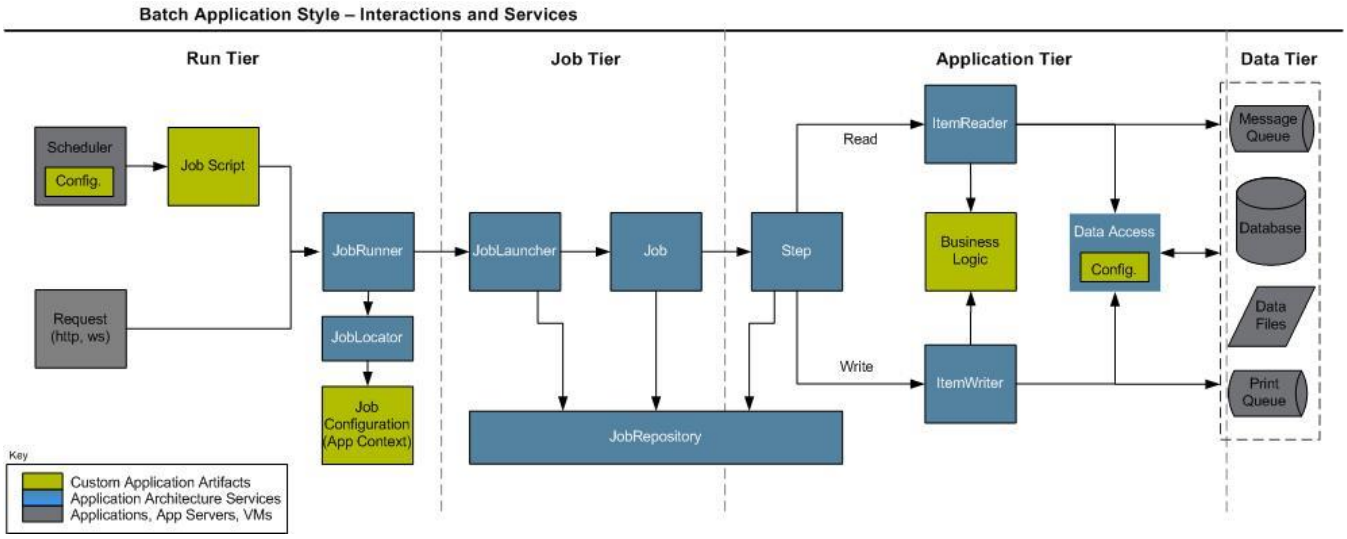
A Job can be re-used to create multiple job instances and this is reflected in the figure above showing an extended picture of the core domain. When a Job is launched it first checks to see if a job with the same *JobParameters* was already executed. We expect one of the following outcomes, depending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

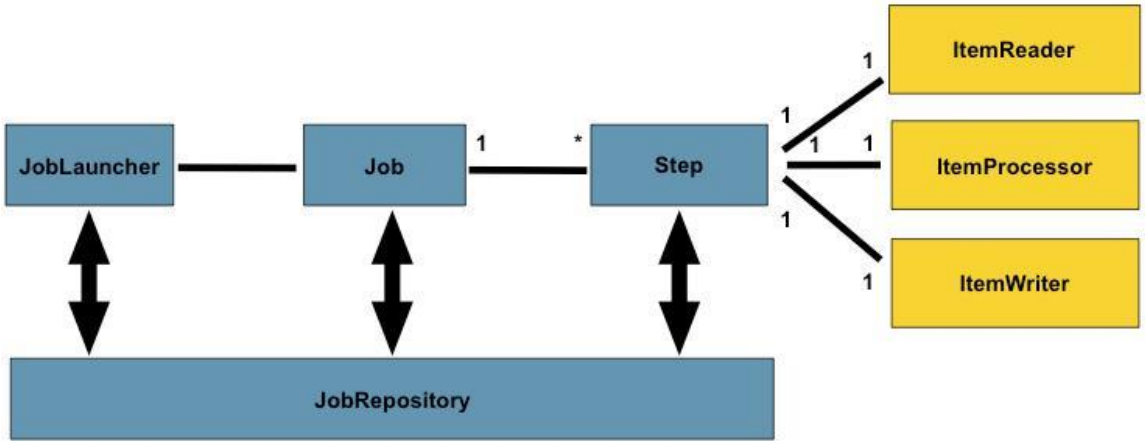


The application style is organized into four logical tiers, which include **Run, Job, Application, and Data** tiers. The primary goal for organizing an application according to the tiers is to embed what is known as "**separation of concerns**" within the system. Effective separation of concerns results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

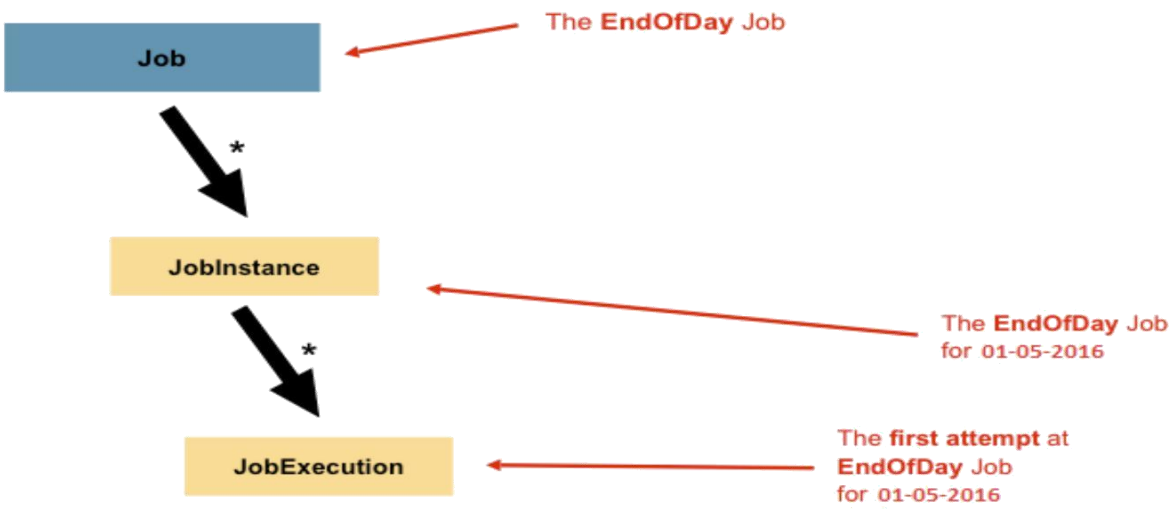
  might include databases, files, or queues. Note: In some cases the Job tier can be completely missing and in other cases one job script can start several batch job instances.

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.

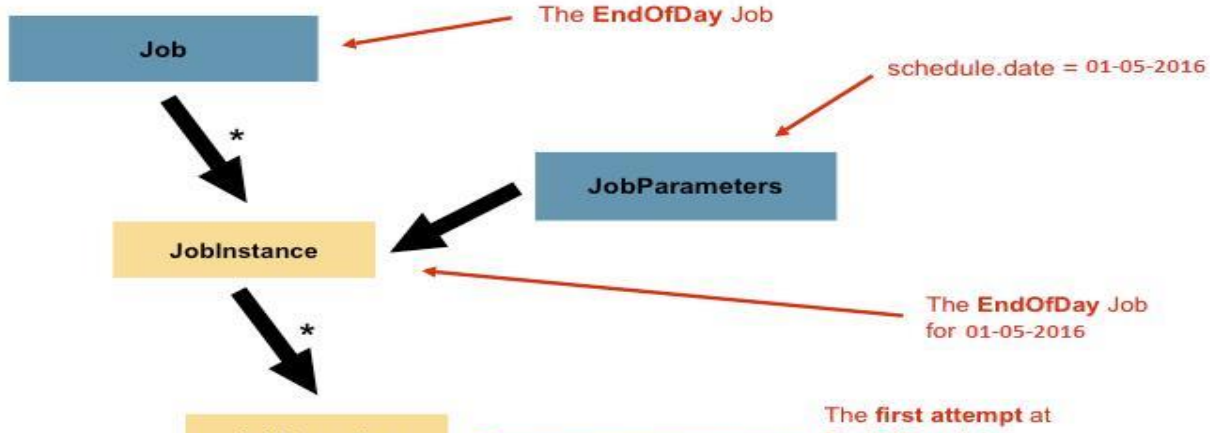
JobParameters

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:



Spring Batch ItemReader and ItemWriter:-

we will discuss about the three most important interfaces of spring batch and an overview of Spring Batch item reader and writer with a sample application. One of the important goals of a batch processing framework is to read large amounts of data, perform some business processing/transformation and write out the result. **Spring Batch Framework** supports this bulk reading, processing and writing using three key interfaces: **ItemReader, ItemProcessor and ItemWriter**.

Popular Spring Tutorials

1. ItemReader is the means for providing data from many different types of input. ItemReader interface is the means for reading bulk data in a bulk processing system. There are many different implementations of ItemReader interface. All implementations are expected to be stateful and will be called multiple times for each batch, with each call to read() returning a different value and finally returning null when all input data is exhausted. Below are few frequently used implementations of ItemReader.

ItemReader Implementation	Description
FlatFileItemReader	Reads lines of data from input file. Typically read line describe records with fields of data defined by fixed positions in the file or delimited by some special character (e.g. Comma (,), Pipe (|) etc).
JdbcCursorItemReader	Opens a JDBC cursor and continually retrieves the next row in the ResultSet.
StoredProcedureItemReader	Executes a stored procedure and then reads the returned

	cursor and continually retrieves the next row in the ResultSet.

All the above implementations override the read() method from the ItemReader interface. The read method defines the most essential contract of the ItemReader. 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 and so on.

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. }
```

2. ItemWriter is similar in functionality to an ItemReader, but with inverse operations. ItemWriter is a interface for generic output operations. Implementation class will be responsible for serializing objects as necessary. Resources still need to be located, opened and

closed but they differ in that an ItemWriter writes out, rather than reading in. For databases these may be inserts or updates.

The write method defines the most essential contract of the ItemWriter. It will attempt to write out the list of items passed in as long as it is open. As it is 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. Once the items are written out , any flushing that may be necessary can be performed before returning from the write method.

ItemWriter Implementation	**Description**
FlatFileItemWriter	Writes data to a file or stream. Uses buffered writer to improve performance.
StaxEventItemWriter	An implementation of ItemWriter which uses StAX and Marshaller for serializing object to XML.

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:

```
public class CompositeItemWriter<T> implements ItemWriter<T>  {

    ItemWriter<T> itemWriter;

    public CompositeItemWriter(ItemWriter<T>  itemWriter) {
        this.itemWriter = itemWriter;
    }

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

        //Add business logic here
       itemWriter.write(item);
    }

    public void setDelegate(ItemWriter<T>  itemWriter){
        this.itemWriter = itemWriter;
   }
}
```

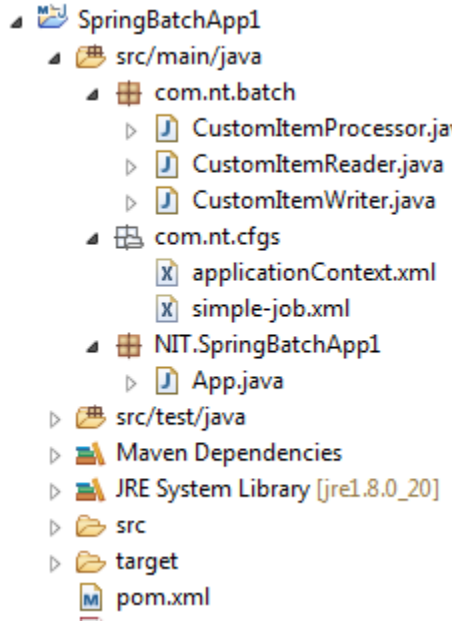
The class above contains another ***ItemWriter*** to which it ***delegates*** after having provided some business logic. This pattern could easily be used for an ***ItemReader*** as well, perhaps to obtain more reference data based upon the input that was provided by the main ***ItemReader***. It is 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:

}

Ex:



CustomItemProcessor.java

```
package com.nt.batch;
import org.springframework.batch.item.ItemProcessor;


publicclass CustomItemProcessor implements ItemProcessor<String, String> {

	@Override
	public String process(String bookNameWithoutAuthor) throws Exception {
		System.out.println("ItemProcessor:process(-)");
		String bookNameWithAuthor = "Book Name - "+bookNameWithoutAuthor+" |
Author Name - ";

		if("Core Java".equalsIgnoreCase(bookNameWithoutAuthor)){
			bookNameWithAuthor += "Sriniwas";
		}elseif("Design Patterns".equalsIgnoreCase(bookNameWithoutAuthor)){
			bookNameWithAuthor += "raja ";
		}elseif("Advance Java".equalsIgnoreCase(bookNameWithoutAuthor)){
```

```
                bookNameWithAuthor += "ravi ";
            }elseif("Spring FrameWork".equalsIgnoreCase(bookNameWithoutAuthor)){
                bookNameWithAuthor += "karan";
            }elseif("Hibernate Framework".equalsIgnoreCase(bookNameWithoutAuthor)){
                bookNameWithAuthor += "rani";
            }
            return bookNameWithAuthor;
        }

}
```

CustomItemReader.java

```
package com.nt.batch;

import java.util.List;
import org.springframework.batch.item.ItemReader;
import org.springframework.batch.item.ParseException;
import org.springframework.batch.item.UnexpectedInputException;

publicclass CustomItemReader implements ItemReader<String>{
        private List<String> bookNameList;
        privateint bookCount = 0;

        @Override
        public String read() throws Exception, UnexpectedInputException,
                        ParseException {
            System.out.println("ItemReader::read()");
            if(bookCount < bookNameList.size()){
                return bookNameList.get(bookCount++);
            }else{
                returnnull;
            }
        }
        /*public List<String> getUserNameList() {
            return bookNameList;
        }*/
        publicvoid setBookNameList(List<String> bookNameList) {
            this.bookNameList = bookNameList;
        }

}
```

CustomItemWriter.java

```
package com.nt.batch;

import java.util.List;
import org.springframework.batch.item.ItemWriter;

publicclass CustomItemWriter implements ItemWriter<String> {

	@Override
	publicvoid write(List<? extends String> bookNameWithAuthor) throws Exception {
		System.out.println("ItemWriter: write(-)");
		System.out.println(bookNameWithAuthor);
	}

}
```

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:context="http://www.springframework.org/schema/context"
	xmlns:p="http://www.springframework.org/schema/p"
	xmlns:mvc="http://www.springframework.org/schema/mvc"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.0.xsd
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd">


	<bean id="transactionManager"
	class="org.springframework.batch.support.transaction.ResourcelessTransactionManage
r"/>

<bean id="jobRepository"

class="org.springframework.batch.core.repository.support.MapJobRepositoryFactoryBean">
<property name="transactionManager" ref="transactionManager"/>
</bean>
```

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

<bean id="simpleJob"
   class="org.springframework.batch.core.job.SimpleJob" abstract="true">
<property name="jobRepository" ref="jobRepository" />
</bean>

</beans>
```

simple-job.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:p="http://www.springframework.org/schema/p"
        xmlns:batch="http://www.springframework.org/schema/batch"
        xmlns:mvc="http://www.springframework.org/schema/mvc"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.0.xsd
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd
http://www.springframework.org/schema/batch
http://www.springframework.org/schema/batch/spring-batch-3.0.xsd">

        <import resource="applicationContext.xml"/>

        <bean id="customReader" class="com.nt.batch.CustomItemReader">
<property name="bookNameList">
<list>
<value>Core Java</value>
<value>Design Patterns</value>
<value>Advance Java</value>
<value>Spring FrameWork</value>
<value>Hibernate Framework</value>
</list>
</property>
</bean>

<bean id="customProcessor" class="com.nt.batch.CustomItemProcessor" />
```

```
<bean id="customWriter" class="com.nt.batch.CustomItemWriter" />

<batch:job id="mySimpleJob" job-repository="jobRepository" parent="simpleJob">
        <batch:step id="step1">
                <batch:tasklet transaction-manager="transactionManager">
                        <batch:chunk reader="customReader" processor="customProcessor"
                                                writer="customWriter" commit-interval="2"/>
                </batch:tasklet>
        </batch:step>
</batch:job>
</beans>
```

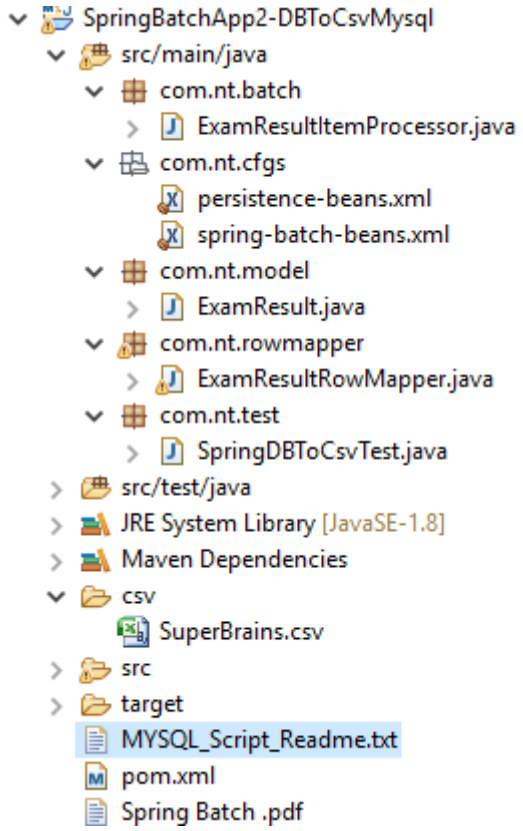
App.java

```
package NIT.SpringBatchApp1;

/**
 * Hello world!
 *
 */
publicclass App
{
publicstaticvoid main( String[] args )
   {
      System.out.println( "Hello World!" );
   }
}
```

pom.xml

```
<project
xmlns="http://maven.apache.org/POM/4.0.0"xmlns:xsi="http://www.w3.org/2001/XMLSchem
a-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>

<groupId>NIT</groupId>
<artifactId>SpringBatchApp1</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<name>SpringBatchApp1</name>
<url>http://maven.apache.org</url>
```

```
<properties>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-context</artifactId>
<version>4.2.2.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework/spring-aop -->
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-aop</artifactId>
<version>4.2.2.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-context-support -->
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-context-support</artifactId>
<version>4.2.2.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework.batch/spring-batch-core -->
<dependency>
<groupId>org.springframework.batch</groupId>
<artifactId>spring-batch-core</artifactId>
<version>3.0.7.RELEASE</version>
</dependency>

</dependencies>
</project>
```



ExamResultItemProcessor.java:-

```
package com.nt.batch;

import org.springframework.batch.item.ItemProcessor;
import com.nt.model.ExamResult;

public class ExamResultItemProcessor implements
ItemProcessor<ExamResult, ExamResult>{


     public ExamResult process(ExamResult result) throws Exception {
          System.out.println("Processing result :"+result);

          /*
           * Only return results which are more than 80%
           *
           */
          if(result.getPercentage() < 80){
               return null;
          }

          return result;
     }
```

```
}
```

persistence-beans.xml:-

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

	<bean id="dbcpDs"
class="org.apache.commons.dbcp2.BasicDataSource">
		<property name="driverClassName"
value="com.mysql.jdbc.Driver" />
		<property name="url"
value="jdbc:mysql://localhost:3306/EXAM_DATA" />
		<property name="username" value="root" />
		<property name="password" value="root" />
	</bean>
</beans>
```

spring-batch-beans.xml:-

```
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:batch="http://www.springframework.org/schema/batch"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/batch
	http://www.springframework.org/schema/batch/spring-batch-3.0.xsd
		http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd">

	<import resource="persistence-beans.xml"/>

	<!-- JobRepository and JobLauncher are configuration/setup
classes -->
	<bean id="jobRepository"

	class="org.springframework.batch.core.repository.support.MapJobRe
positoryFactoryBean" />

	<bean id="jobLauncher"

	class="org.springframework.batch.core.launch.support.SimpleJobLau
ncher">
		<property name="jobRepository" ref="jobRepository" />
```

```
	</bean>



	<!-- ItemReader which reads from database and returns the row
mapped by
		rowMapper -->
	<bean id="dbItemReader"

	class="org.springframework.batch.item.database.JdbcCursorItemRead
er">
		<property name="dataSource" ref="dbcpDs" />
		<property name="sql"
			value="SELECT ID,SEMESTER, DOB, PERCENTAGE FROM
EXAM_RESULT" />
		<property name="rowMapper">
		   <bean class="com.nt.rowmapper.ExamResultRowMapper"/>
		</property>
	</bean>

	<!-- ItemWriter writes a line into output CSV file -->
	<bean id="flatFileItemWriter"
class="org.springframework.batch.item.file.FlatFileItemWriter">
		<property name="resource" value="file:csv/SuperBrains.csv"
/>
		<property name="lineAggregator">
			<!-- An Aggregator which converts an object into
delimited list of strings -->
			<bean
	class="org.springframework.batch.item.file.transform.DelimitedLin
eAggregator">
				<property name="delimiter" value="," />
				<property name="fieldExtractor">
					<!-- Extractor which returns the value of
beans property through reflection -->
					<bean
	class="org.springframework.batch.item.file.transform.BeanWrapperF
ieldExtractor">
						<property name="names" value="id, sem,
percentage, dob" />
					</bean>
				</property>
			</bean>
		</property>
	</bean>
```

```
	<!-- Optional ItemProcessor to perform business logic/filtering
on the input
		records -->
	<bean id="itemProcessor"
class="com.nt.batch.ExamResultItemProcessor" />


	<!-- Step will need a transaction manager -->
	<bean id="transactionManager"

	class="org.springframework.batch.support.transaction.Resourceless
TransactionManager" />

	<!-- Actual Job -->
	<batch:job id="examResultJob">
		<batch:step id="step1">
			<batch:tasklet transaction-
manager="transactionManager">
				<batch:chunk reader="dbItemReader"
writer="flatFileItemWriter"
					processor="itemProcessor" commit-
interval="2" />
			</batch:tasklet>
		</batch:step>
	</batch:job>

</beans>
```

ExamResult.java:-

```
package com.nt.model;

import java.sql.Date;

public class ExamResult {
	private int id;
	private int sem;
	private Date dob;
	private double percentage;


	public int getId() {
		return id;
	}

	public void setId(int id) {
```

```
		this.id = id;
	}

	public int getSem() {
		return sem;
	}

	public void setSem(int sem) {
		this.sem = sem;
	}

	public Date getDob() {
		return dob;
	}

	public void setDob(Date dob) {
		this.dob = dob;
	}

	public double getPercentage() {
		return percentage;
	}

	public void setPercentage(double percentage) {
		this.percentage = percentage;
	}

	@Override
	public String toString() {
		return "ExamResult [id=" + id + ", sem=" + sem + ", dob=" +
dob + ", percentage=" + percentage + "]";
	}

}
```

ExamResultRowMapper.java:-

```
package com.nt.rowmapper;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;
import com.nt.model.ExamResult;

public class ExamResultRowMapper implements RowMapper<ExamResult>{
```

```
	public ExamResult mapRow(ResultSet rs, int rowNum) throws
SQLException {
		ExamResult result =null;
		result=new ExamResult();
		result.setId(rs.getInt("id"));
		result.setSem(rs.getInt("Semester"));
		result.setDob(rs.getDate("dob"));
		result.setPercentage(rs.getDouble("percentage"));
		return result;
	}

}
```

SpringDBToCsvTest.java:-

```
package com.nt.test;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.JobExecutionException;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class SpringDBToCsvTest {

	@SuppressWarnings("resource")
	public static void main(String areg[]){
		ApplicationContext ctx =null;
		JobLauncher jobLauncher=null;
		JobExecution execution=null;
		Job job=null;

		ctx=new ClassPathXmlApplicationContext("com/nt/cfgs/spring-
batch-beans.xml");

		jobLauncher = (JobLauncher) ctx.getBean("jobLauncher");
		job = (Job) ctx.getBean("examResultJob");
		try {
			execution = jobLauncher.run(job, new
JobParameters());
			System.out.println("Job Exit Status : "+
execution.getStatus());
		} catch (JobExecutionException e) {
```

```
                System.out.println("Job ExamResult failed");
                e.printStackTrace();
            }
      }//main
}//class
```

pom.xml:-

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>

      <groupId>nit</groupId>
      <artifactId>SpringBatchDatabaseToCsv</artifactId>
      <version>1.0</version>
      <packaging>jar</packaging>

      <name>SpringBatchDatabaseToCsv</name>
      <url>http://maven.apache.org</url>

      <dependencies>

            <!--
https://mvnrepository.com/artifact/org.springframework/spring-context
-->
            <dependency>
                  <groupId>org.springframework</groupId>
                  <artifactId>spring-context</artifactId>
                  <version>4.3.9.RELEASE</version>
            </dependency>

            <!--
https://mvnrepository.com/artifact/org.springframework/spring-context-
support -->
            <dependency>
                  <groupId>org.springframework</groupId>
                  <artifactId>spring-context-support</artifactId>
                  <version>4.3.9.RELEASE</version>
            </dependency>

            <dependency>
                  <groupId>org.springframework</groupId>
                  <artifactId>spring-core</artifactId>
                  <version>4.3.9.RELEASE</version>
```

```
            </dependency>
            <dependency>
                  <groupId>org.springframework</groupId>
                  <artifactId>spring-jdbc</artifactId>
                  <version>4.3.9.RELEASE</version>
            </dependency>

            <dependency>
                  <groupId>org.springframework.batch</groupId>
                  <artifactId>spring-batch-core</artifactId>
                  <version>3.0.8.RELEASE</version>
            </dependency>
            <dependency>
                  <groupId>org.springframework.batch</groupId>
                  <artifactId>spring-batch-infrastructure</artifactId>
                  <version>3.0.8.RELEASE</version>
            </dependency>

             <!--
https://mvnrepository.com/artifact/org.apache.commons/commons-dbcp2 --
>
            <dependency>
                  <groupId>org.apache.commons</groupId>
                  <artifactId>commons-dbcp2</artifactId>
                  <version>2.1.1</version>
            </dependency>


            <dependency>
                  <groupId>mysql</groupId>
                  <artifactId>mysql-connector-java</artifactId>
                  <version>5.1.38</version>
            </dependency>
      </dependencies>
</project>
```

How to create Procedure in MySql DataBase:-

MYSQL_Script_Readme.txt:-

```
Step-1(Create Database)
===============
create Database EXAM_DATA;

Step-2(Use Database)
===============
```

```
use EXAM_DATA;


Step-3 (Create Table)
=====================
CREATE TABLE `EXAM_RESULT`
(
  `id`                bigint(20) NOT NULL      AUTO_INCREMENT,
  `dob`               timestamp  NOT NULL      DEFAULT
CURRENT_TIMESTAMP,
  `Semester`     int(11)                  DEFAULT NULL,
  `percentage`       float                    DEFAULT NULL,

  PRIMARY KEY (`id`)
);


Step-4 (Create Procedure to Insert )
=====================================
DELIMITER $$
CREATE PROCEDURE generate_EXAM_RESULT()
BEGIN
  DECLARE i INT DEFAULT 0;


WHILE i < 500000 DO
    INSERT INTO `EXAM_RESULT` (`dob`,`percentage`,`Semester`) VALUES (
      FROM_UNIXTIME(UNIX_TIMESTAMP('2000-01-01
01:00:00')+FLOOR(RAND()*31536000)),
      ROUND(RAND()*100,2),
      1
    );
    SET i = i + 1;
  END WHILE;
END$$
DELIMITER ;

Step-5 (Call Procedure to Insert 50K Data )=> Takes Least 30-45 Min to
Insert the records
=========================================================================
====================
CALL generate_EXAM_RESULT();

Step-6 (Check the Records )
===================================
select * from EXAM_RESULT;
```

```
Step-7 /Optional (Remove Procedure if you dont need it any more )
=====================================================================
DROP PROCEDURE generate_EXAM_RESULT;
```