Spring Batch:

* Spring Batch is one of the core modules of spring framework and using this spring batch you can create robust processing system.
* It is used for high volume batch processing.
* Spring Batch is a processing framework designed for robust execution of jobs.

What is Batch Processing?

* Batch processing is a technique which processes data in a large group instead of a single element of data where you can process a high volume of data with minimal human interaction.

When to use Batch processing?

* You can use Spring Batch in both simple use cases (such as reading a file into a database or running a stored procedure) and complex, high volume use cases (such as moving high volumes of data between databases, transforming it, and so on).

Spring Batch core component & architecture flow:

1. Job Launcher:

* It’s an interface and an entry point to call any job in Spring Batch.
* It has a method called run() which triggers the job component.

1. Job:

* A job can be defined as the work to be executed by the Spring Batch.

1. Job Repository:

* Once job launcher launches job, immediately it calls another component called Job Repository
* It is used to maintain state of the job whether it is success or failure.

1. Step:

* A Job is further divided into steps.
* A **step** is an independent part of a job which contains the necessary information to define and execute the job (its part).
* As specified in the diagram, each step is composed of an ItemReader, ItemProcessor (optional) and an ItemWriter. **A job may contain one or more steps**.
* For example, In a bank transaction there may involve multiple steps as shown

Communicate with other Bank

Debit Money

Transaction

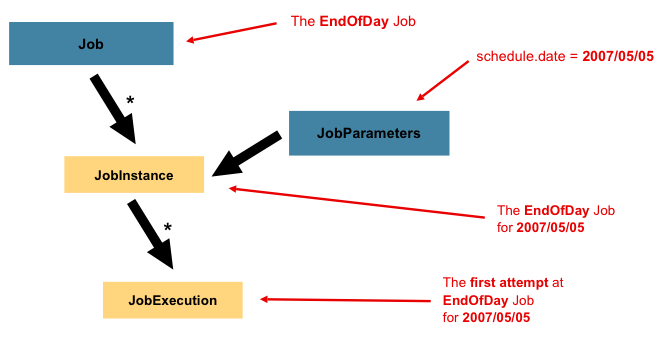
Job

Send SMS/Email

Steps

1. Readers, Writers, and Processors:

* An **item reader** reads data into a Spring Batch application from a particular source, whereas an **item writer** writes data from the Spring Batch application to a particular destination.
* An **Item processor** is a class which contains the processing code which processes the data read into the spring batch. If the application reads **"n"** records, then the code in the processor will be executed on each record.

**Job Hierarchy:** A Job is an entity that encapsulates an entire batch process. As is common with other Spring projects, a Job is wired together with either an XML configuration file or Java-based configuration. This configuration may be referred to as the “job configuration”. However, Job is only the top of an overall hierarchy, as shown in the following diagram: 

In Spring Batch, a Job is simply a container for Step instances. It combines multiple steps that logically belong together in a flow and allows for configuration of properties global to all steps, such as restartability. The job configuration contains:

* The name of the job.
* Definition and ordering of Step instances.
* Whether or not the job is restartable.

**Job Instance:**

* A JobInstance refers to the concept of a logical job run.
* Consider a batch job that should be run once at the end of the day, such as the EndOfDay Job from the preceding diagram. There is one EndOfDay job, but each individual run of the Job must be tracked separately. In the case of this job, there is one logical JobInstance per day.
* For example, there is a January 1st run, a January 2nd run, and so on. If the January 1st run fails the first time and is run again the next day, it is still the January 1st run.

**Job Parameters:**

* Having discussed JobInstance and how it differs from Job, the natural question to ask is: “How is one JobInstance distinguished from another?” The answer is: JobParameters.
* **A JobParameters object holds a set of parameters used to start a batch job**. They can be used for identification or even as reference data during the run.
* In the preceding example, where there are two instances, one for January 1st and another for January 2nd, there is really only one Job, but it has two JobParameter objects: one that was started with a job parameter of 01-01-2017 and another that was started with a parameter of 01-02-2017.
* Thus, the contract can be defined as: **JobInstance = Job + identifying JobParameters.**

**Job Execution:**

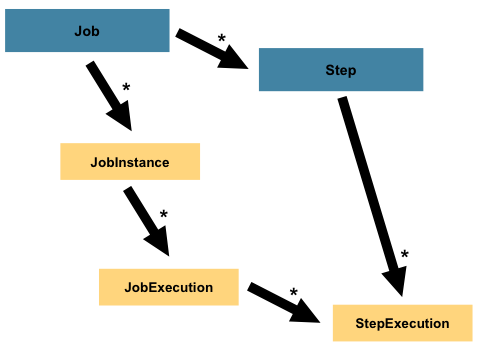
* A JobExecution refers to the technical concept of a single attempt to run a Job. An execution may end in failure or success, but the JobInstance corresponding to a given execution is not considered to be complete unless the execution completes successfully.
* Using the EndOfDay Job described previously as an example, consider a JobInstance for 01-01-2017 that failed the first time it was run. If it is run again with the same identifying job parameters as the first run (01-01-2017), a new JobExecution is created. However, there is still only one JobInstance.
* Each time JobInstance fails, a new JobExecution is created but JobInstance still remains same.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***JobExecution Properties:***   |  |  | | --- | --- | | Property | Definition | | Status | A BatchStatus object that indicates the status of the execution. While running, it is BatchStatus#STARTED. If it fails, it is BatchStatus#FAILED. If it finishes successfully, it is BatchStatus#COMPLETED | | startTime | A java.util.Date representing the current system time when the execution was started. This field is empty if the job has yet to start. | | endTime | A java.util.Date representing the current system time when the execution finished, regardless of whether or not it was successful. The field is empty if the job has yet to finish. | | exitStatus | The ExitStatus, indicating the result of the run. It is most important, because it contains an exit code that is returned to the caller. See chapter 5 for more details. The field is empty if the job has yet to finish. | | 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 always has 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. This field is empty if the job has yet to start. | | 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. | | |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

* These properties are important because they are persisted and can be used to completely determine the status of an execution.

**Step Hierarchy:**

 A Step contains all of the information necessary to define and control the actual batch processing.



#### StepExecution:

* A StepExecution represents a single attempt to execute a Step.
* A new StepExecution is created each time a Step is run, similar to JobExecution. However, if a step fails to execute because the step before it fails, no execution is persisted for it. A StepExecution is created only 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 counts and start and end times.
* The following table lists the properties for StepExecution:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **StepExecution Properties:**   |  |  | | --- | --- | | **Property** | **Definition** | | Status | A BatchStatus object that indicates the status of the execution. While running, the status is BatchStatus.STARTED. If it fails, the status is BatchStatus.FAILED. If it finishes successfully, the status is BatchStatus.COMPLETED. | | startTime | A java.util.Date representing the current system time when the execution was started. This field is empty if the step has yet to start. | | endTime | A java.util.Date representing the current system time when the execution finished, regardless of whether or not it was successful. This field is empty if the step has yet to exit. | | exitStatus | The ExitStatus indicating the result of the execution. It is most important, because it contains an exit code that is returned to the caller. See chapter 5 for more details. This field is empty if the job has yet to exit. | | 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 of 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. | |
|  |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Task Executor:** **An object that executes submitted Runnable tasks**.

* This intesrface provides a way of decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling, etc.

**SimpleAsyncTaskExecutor :** This starts a new thread and executes it asynchronously. It does not reuse the thread.

@Bean  
 public TaskExecutor taskExecutor() {  
 SimpleAsyncTaskExecutor asyncTaskExecutor = new SimpleAsyncTaskExecutor();  
 asyncTaskExecutor.setConcurrencyLimit(10);  
 return asyncTaskExecutor;  
 }

**Stopping the job:** We can stop the job which is executing using JobOperator & Job Execution Id as path variable. But Immediately Spring Batch does not get stopped, it proceeds with the current chunk & stops.

@Autowired  
JobOperator jobOperator;  
  
@GetMapping("/stop/{jobExecutionId}")  
public String stopJob(@PathVariable long jobExecutionId) {  
 try {  
 jobOperator.stop(jobExecutionId);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 return "Job Stopped...";  
}

What is partitioning?

* Multiple threads to process a range of data sets.
* Suppose, if we have 1000 records in data set
  + We can give 1-500 to thread1 &s
  + 501-1000 to thread2
  + So, each thread gets a range/chunk of 500 records

**Spring Batch Listeners:**

* Spring Batch listeners are a way of intercepting the execution of a Job or a Step to perform some meaningful operations or logging the progress.
* We have the following types of event listeners which intercept the batch processing at specific events.
* JobExecutionListener (before and after job)
* StepExecutionListener (before and after step)
* ChunkListener
* ItemReadListener
* ItemProcessListener
* ItemWriteListener
* SkipListener

**Types Of Steps:**There are two type of steps, namely tasklet & chunk oriented

1. **Tasklets:** Tasklets are meant to perform a single task within a step. Our job will consist of several steps that execute one after the other. **Each step should perform only one defined task**.

Our job will consist of three steps:

1. Read lines from the input CSV file.
2. Calculate age for every person in the input CSV file.
3. Write name and age of each person to a new output CSV file.
4. Chunk Oriented Step:As the name suggests, this approach **performs actions over chunks of data**. That is, instead of reading, processing and writing all the lines at once, it'll read, process and write a fixed amount of records (chunk) at a time.

Then, it'll repeat the cycle until there's no more data in the file.

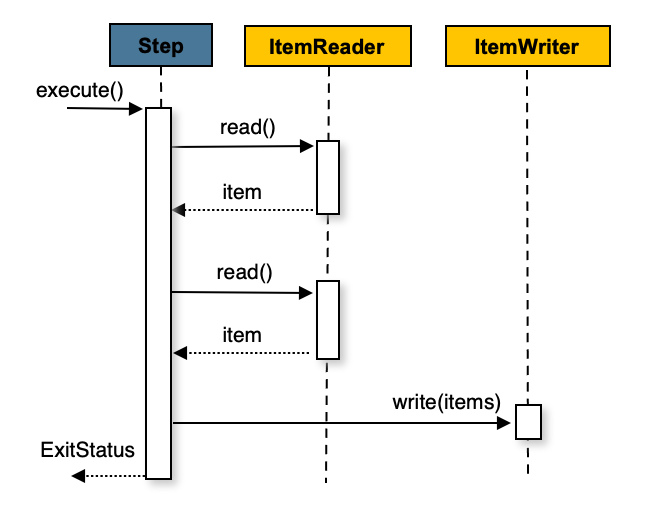
As a result, the flow will be slightly different:

While there're lines:

1. Do for X amount of lines:
   1. Read one line
   2. Process one line
2. Write X amount of lines.

| **Comparison of chunk model and tasklet model:** | | |
| --- | --- | --- |
| **Item** | **Chunk** | **Tasklet** |
| Components | It consists of 3 components mainly ItemReader, ItemProcessor and ItemWriter. | It is consolidated in one Tasklet. |
| Transaction | A certain number of records are processed by issuing intermediate commit. Batch commit cannot be done. It can be processed by specific machine resources regardless of the data count. If an error occurs in the midway, then unprocessed data and processed data will get mixed. | The data is entirely processed by batch commit. There is a need for the user to implement intermediate commit. If the data to be processed is large, machine resources may get exhausted. If an error occurs in the midway, only the unprocessed data is rolled back. |
| Restart | It can be restarted based on the record count. | It cannot be restarted based on the record count. |

Spring Batch uses a “chunk-oriented” processing style in its most common implementation. Chunk oriented processing refers to reading the data one at a time and creating 'chunks' that are written out within a transaction boundary. Once the number of items read equals the commit interval, the entire chunk is written out by the ItemWriter, and then the transaction is committed. The following image shows the process:



**The following pseudo code shows the same concepts in a simplified form:**

List items = new Arraylist();

for(int i = 0; i < commitInterval; i++){

Object item = itemReader.read();

if (item != null) {

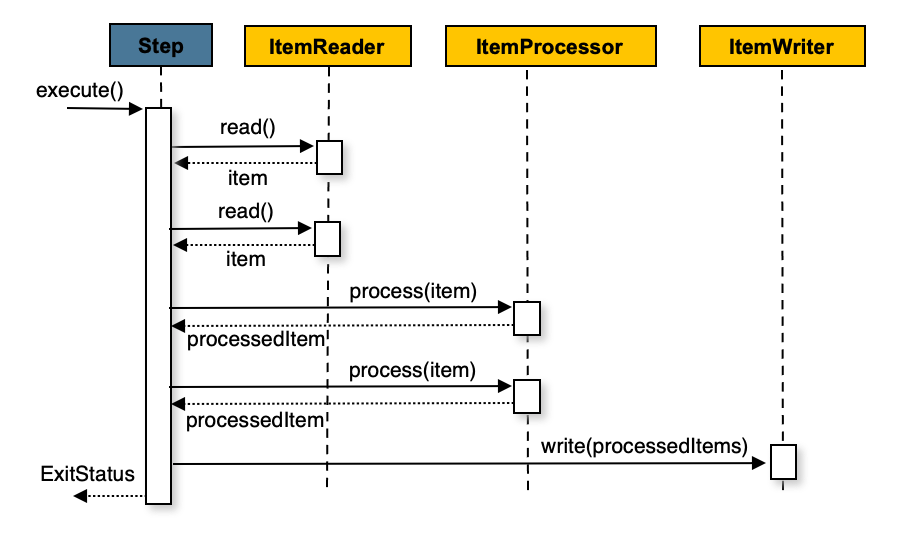
items.add(item);

}

}

itemWriter.write(items);

You can also configure a chunk-oriented step with an optional ItemProcessor to process items before passing them to the ItemWriter. The following image shows the process when an ItemProcessor is registered in the step:



**The following pseudo code shows how this is implemented in a simplified form:**

List items = new Arraylist();

for(int i = 0; i < commitInterval; i++){

Object item = itemReader.read();

if (item != null) {

items.add(item);

}

}

List processedItems = new Arraylist();

for(Object item: items){

Object processedItem = itemProcessor.process(item);

if (processedItem != null) {

processedItems.add(processedItem);

}

}

itemWriter.write(processedItems);

**Spring Batch Fault Tolerance:**

The components ItemReader, ItemProcessor and ItemWritter will throw an exceptions in case of error. To avoid this fault tolerance should be done using **Skip Policy & Retry Mechanism.**

**Using Skip Policy:**

* If an error occurs in one row while processing data set, entire data set transaction is rolled back & Batch job fails.
* We need to define skip policy while building step object.
* Listener to log details of items skipped. Always keep listener before skip policy while configuring in step object.
* Suppose we have got skipped 3 records due to exception and you have seen that from listener & identified which row are causing error & corrected them. If you rerun the application then Spring Batch only inserts those 3 records to the DB & does not scan rest of the rows.

@Bean  
 public Step step1() {  
 return stepBuilderFactory.get("csv-step").<Customer, Customer>chunk(10)  
 .reader(reader())  
 .processor(processor())  
 .writer(writer())  
 .faultTolerant()  
 .skipLimit(100)  
 .skip(DataFormatException.class)  
 .noSkip(IllegalArgumentException.class)  
 .taskExecutor(taskExecutor())  
 .build();  
 }

**Spring Batch Dynamic file upload:**

@StepScope: The step scope means that Spring will create the bean only when the step asks for it and that values will be resolved then (this is the lazy instantiation pattern; the bean isn't created during the Spring application context's bootstrapping).

Marshalling & UnMarshalling:

Configuring Multiple DB’s:

@Primary & @ConfigurationProperties: