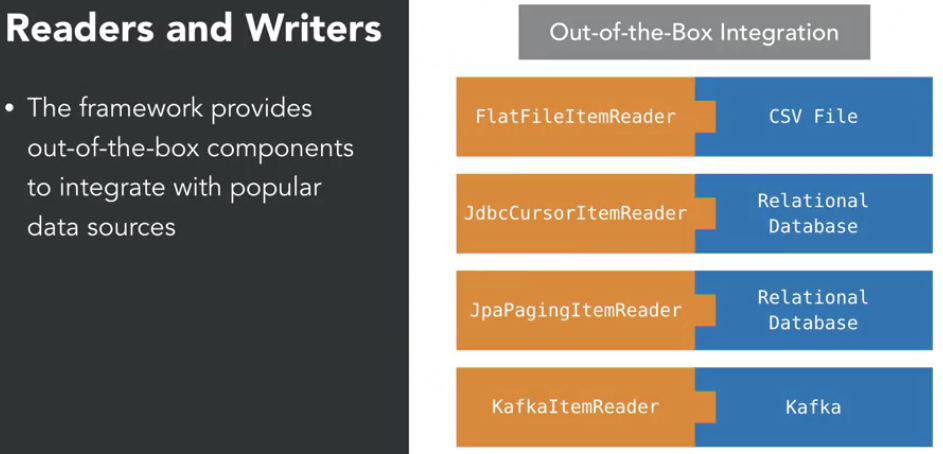
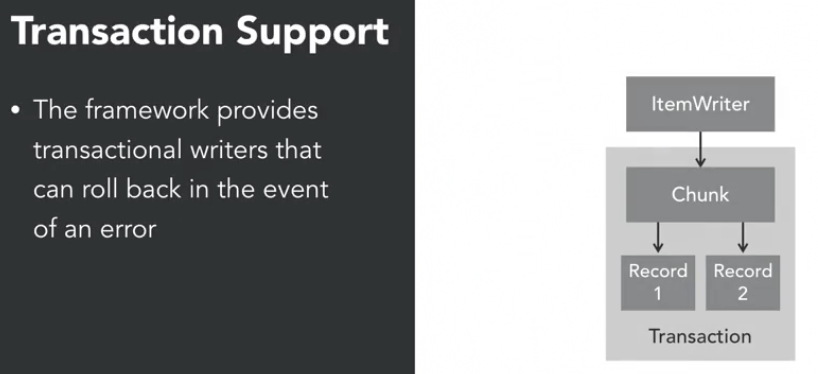
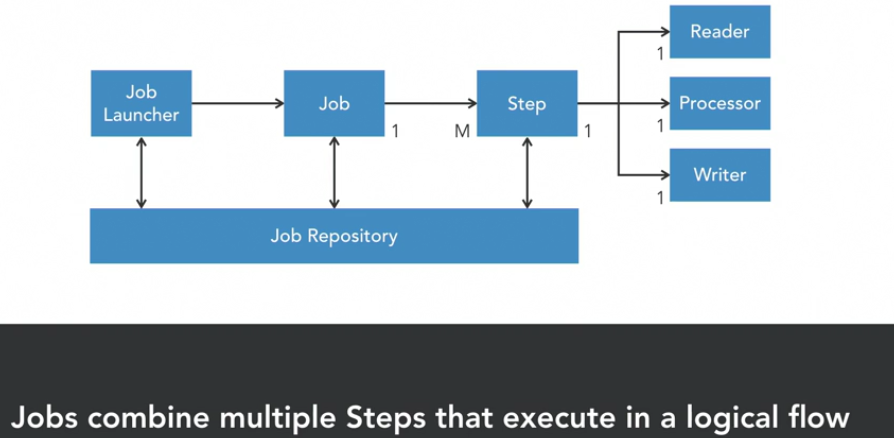




 Out of the box Spring Batch provides the ability to capture the state of our batch jobs as they execute. During execution of a job, Spring Batch writes metadata to a job repository at various points in time, we don't even need to write any code to make this happen. This is very helpful when determining what jobs have executed, why a job has failed, and is used to support additional batch functionality like restarts. In the event a job does fail Spring Batch provides the capability to restart the job from where we left off based upon information about the job in the job repository. So if we execute this job and processing fails when reading the second chuck of data in step two, the entire job will be marked as failed. So we see step two marked as failed, and then our job ultimately has failed. We can then use the framework to relaunch the job. And the job will begin to execute, skipping those steps that have executed successfully and starting at our failed step, in this case step two. Step two will not process the first chunk of data, it's going to restart at the second chuck, and here we see that it completes successfully. At that point, both our step and job will be marked as successfully executed within the job repository. Most batch jobs need to read data from some data source, do some processing, then write the process data to another data source. Spring Batch provides the item reader and item writer interfaces to abstract this concept and provides out of the box implementations for consuming or writing data to popular data sources like flat files, relational databases, XML files, JSON files, and Kafka. So the framework handles a lot of the heavy lifting when it comes to reading and writing data in a batch job.



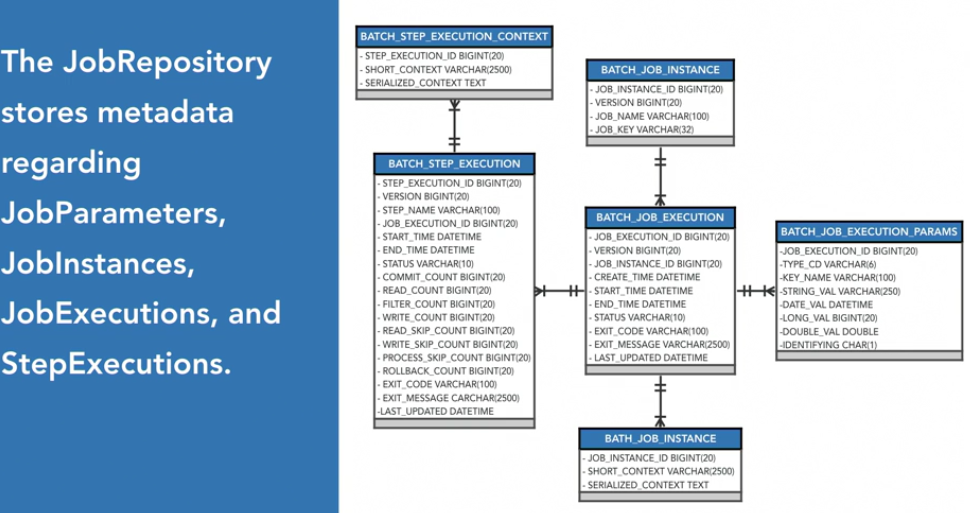


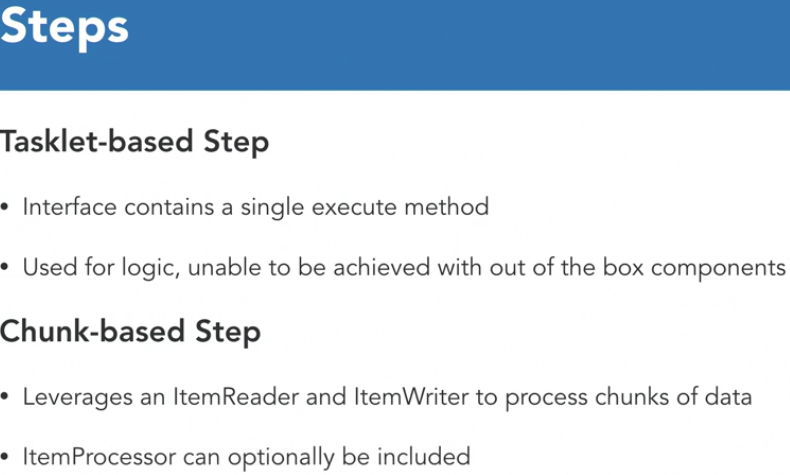


A job represents the entire batch process that we want to execute. It defines one or more steps that execute in an order we commonly call a flow. A step is a phase in a batch job that defines how the actual processing will occur for that portion of the job. The processing logic within a step may read data from a data source, process it and then write it to another data source. A job can contain multiple steps and the flow from one step to another can be dynamic. Meaning it can be conditional or occur in parallel. The entire job is launched using a Job Launcher, which may pass JobParameters to the job.



Each time we execute a jobInstance, we get new job Execution.





Add below line in application.properties to configure the Job repository tables in database –

spring.batch.initialize-schema=always

in new spring boot 2.5.4 and above, this property key is changed as

spring.batch.jdbc.initialize-schema=always

when application starts, spring batch loads the tables mentioned below to the schema specified if it doesn’t exists automatically.

select \* from BATCH\_JOB\_EXECUTION;

select \* from BATCH\_JOB\_EXECUTION\_CONTEXT;

select \* from BATCH\_JOB\_EXECUTION\_PARAMS;

select \* from BATCH\_JOB\_INSTANCE;

select \* from BATCH\_STEP\_EXECUTION;

select \* from BATCH\_STEP\_EXECUTION\_CONTEXT;

Below sequences are also added -

BATCH\_JOB\_EXECUTION\_SEQ;

BATCH\_JOB\_SEQ;

BATCH\_STEP\_EXECUTION\_SEQ;

**Job Parameters -**

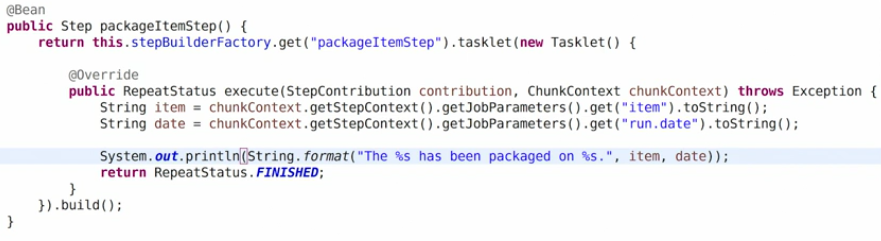
 Job instances are created using the name of the job and parameters passed by a job launcher. If a job instance has been successfully executed, it's not possible to rerun that same job instance. Instead, a new job instance must be created by passing a new job parameter to the job.

Job parameters can be provided while running the jar file from command line as follows –

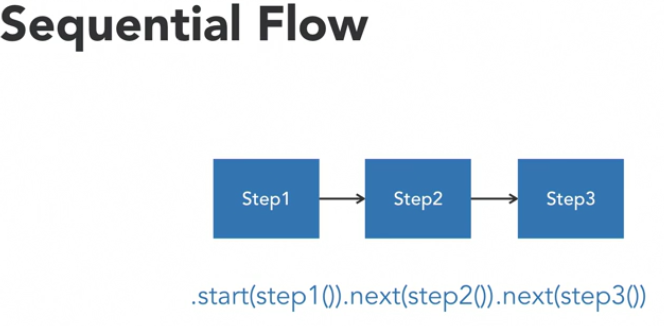
Java -jar app1.jar “items=shoes” “run.date(date)=2020/01/01”

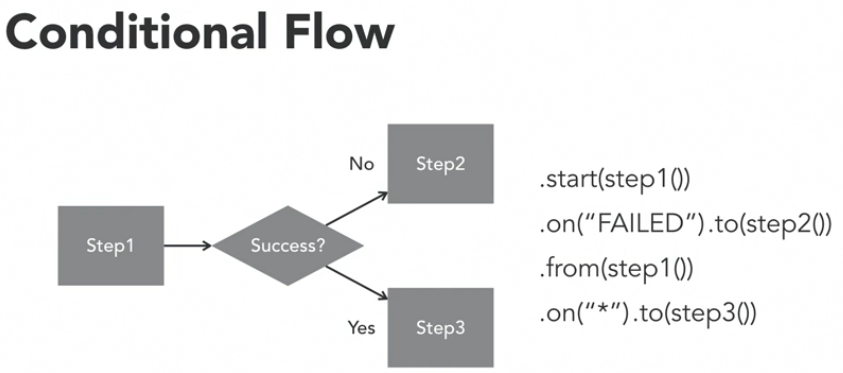
Items and run.date should be read using chunk context

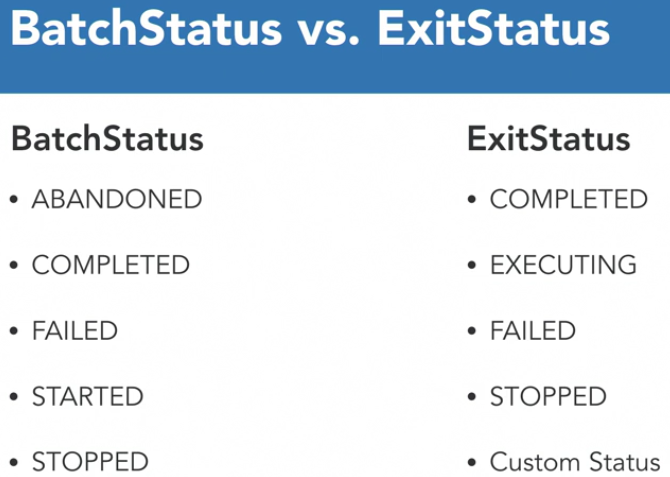
Date format is in YYYY/DD/MM format



Spring Batch allows failed jobs to be restarted. The framework only allows a job to be restarted if the overall status of a job execution is marked as **failed or stopped**. By default, it does not allow completed jobs to be restarted. When a job is restarted, Spring Batch will create a new job execution for the particular job instance that failed, and it will restart at the failed step, executing from that point forward.

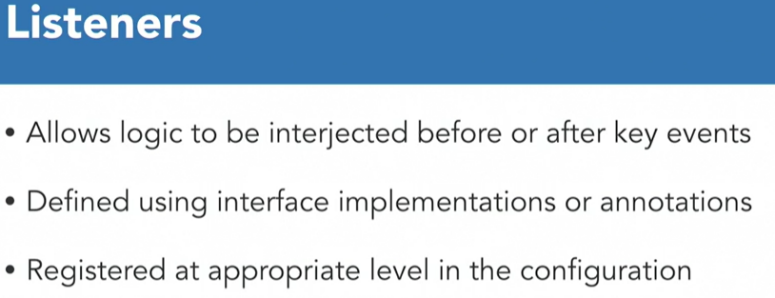
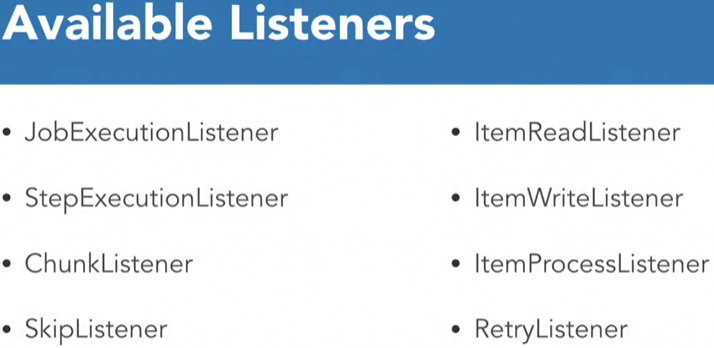


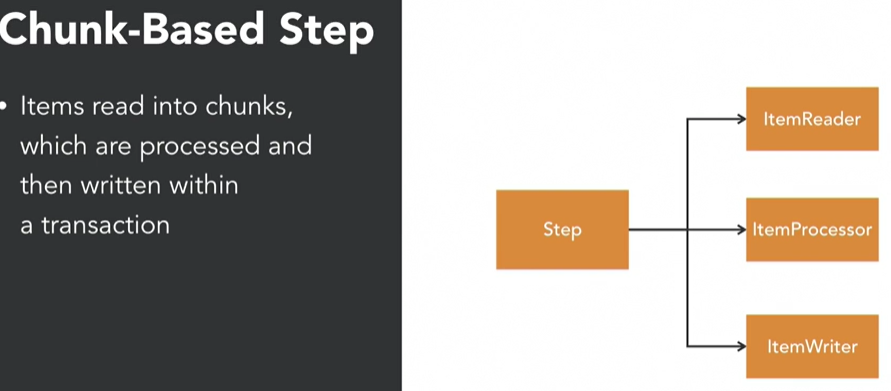




The JobExecution decider is registered within the Job configuration and is used to provide a custom ExitStatus following the execution of a step

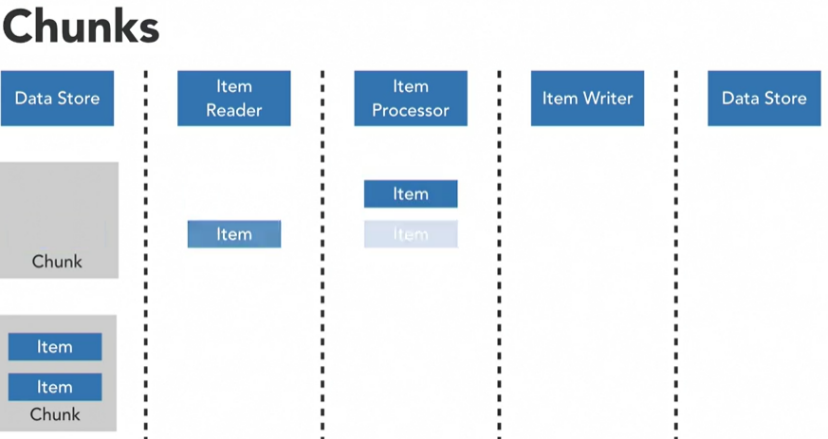
In collaboration with transitions the ExitStatus of step is matched against a pattern to define conditional flows

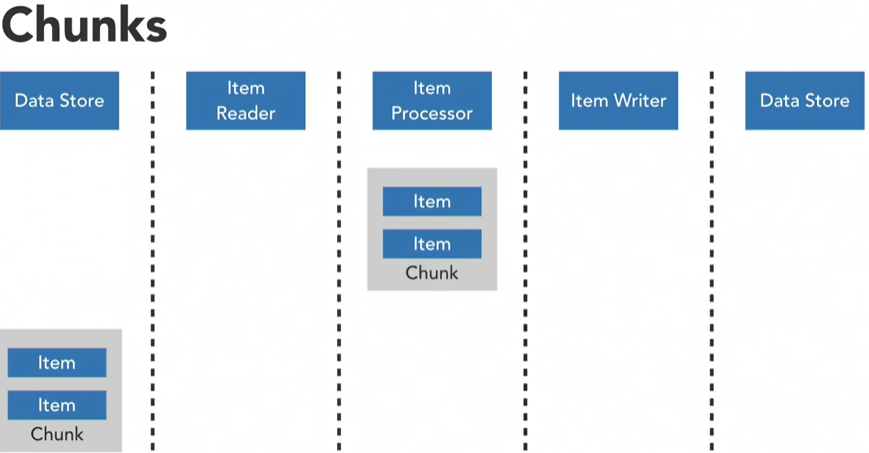


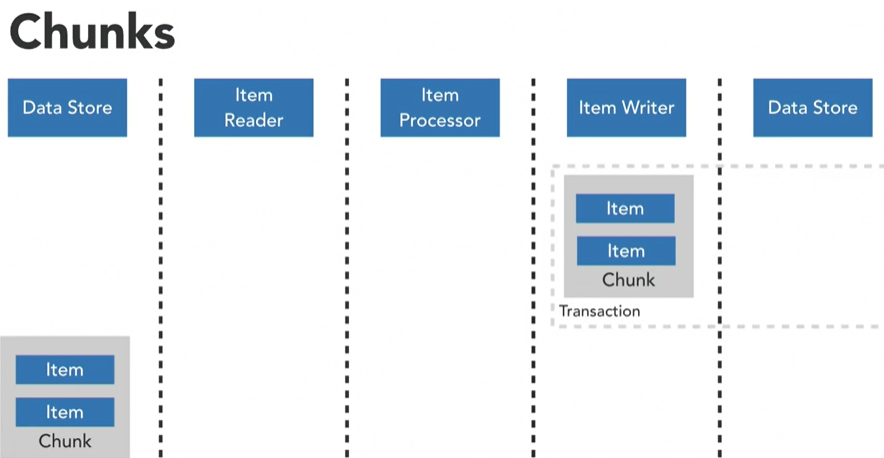


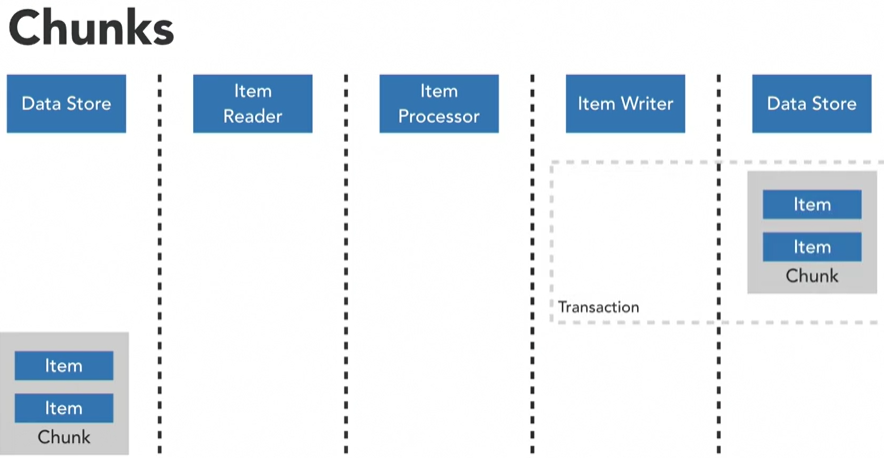
 When performing a chunk-based step, we typically provide a chunk size which determines how many items will be found within a chunk. In this example, our chunk size has been set to two, so you'll notice that each of the chunks contains two items. When processing starts, the ItemReader will read the first item in the chunk and then pass it to the processor for processing. It then repeats this process for the next item within the chunk. Once we've met the chunk size, the entire chunk will be passed to the ItemWriter and then will be written to a data store within a transaction. Once the chunk has been written, this process will repeat itself, will read the first item in the chunk, then process it, then read the second item the chunk and process that one, and then we're going to take the entire chunk and use the ItemWriter to write it to a data store.

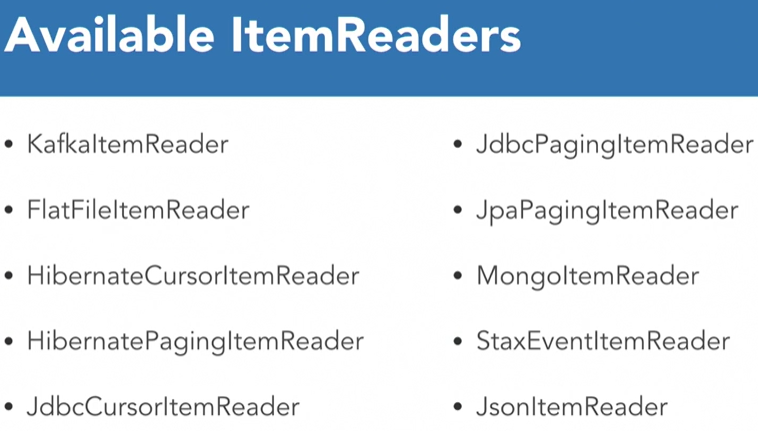




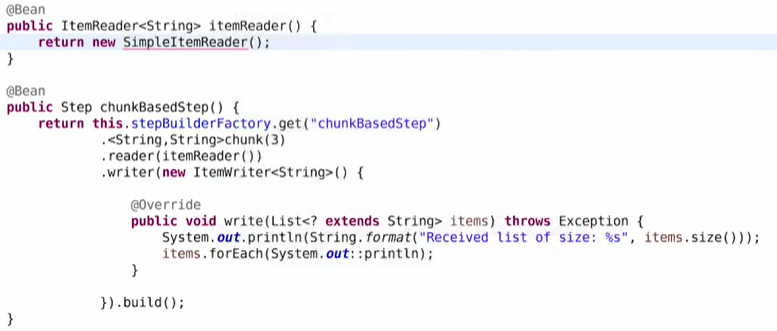




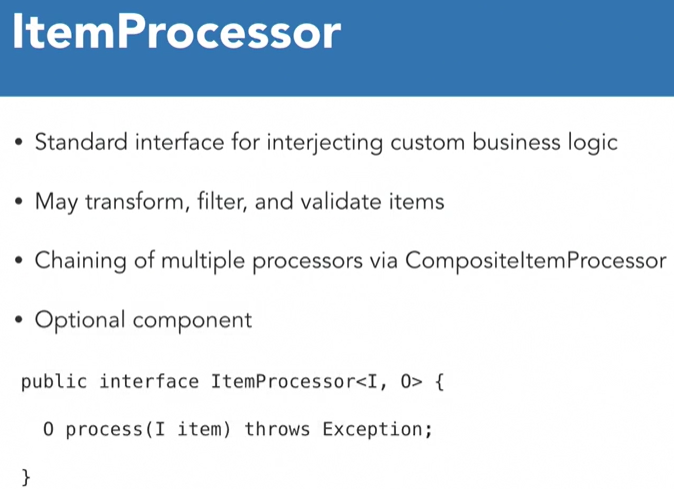










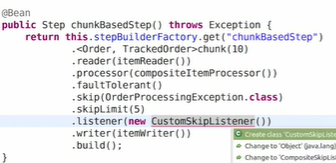


I stands for type of Input, O stand for type of output

A CompositeItemProcessor allows multiple ItemProcessors to be chained together and executed in sequence.

**Skips**

 Skips allow us to continue processing when a particular item causes an exception to be thrown. The job will just ignore the exception and continue processing. So let's imagine that we permit skips in this job. You will notice that processing of our second item is going to fail. However, our job is going to continue on processing, and we will able to complete the batch. Skips are a great way to make batch jobs more resilient in non-critical jobs.

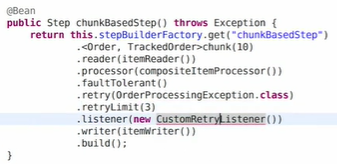


We can configure the skip for type of exception.

skip limit indicates how many skips can occur within this step before we would actually fail the job. Because if every item within this step were to throw that exception, there's probably something a little bit more critical going on, and we wouldn't want to fail.

**Retry -**

Let's imagine we kick off a job and that we must call a web service to process items within our data set. During processing of the second item in the chunk, the web service has an intermittent issue. It's not a hard failure. It could just be small network blip. If we have configured Spring Batch's retry capabilities, the framework will reattempt processing of the item. This time when we attempt to process the item, the web service has become available. So, when we reattempt processing, the item will complete successfully without any intervention by a batch job operator. Our job then can successfully complete without anyone losing sleep.



**Multi-threaded steps**

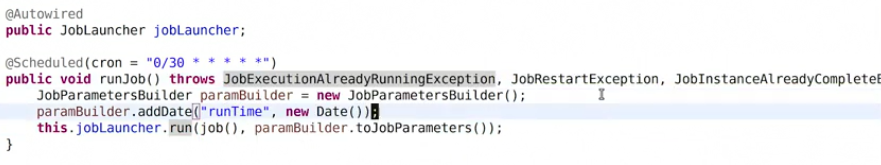
we can make use of thread pool executor.

Scheduling jobs –



Spring boot scheduling –

1. Add @EnableScheduling annotation
2. Add spring.batch.job.enabled=false in application.properties . This will disable the job from starting immediately on launch.
3. Create a Job launcher and a method with @Scheduled to perform scheduling



Above cron expression causes the job to be executed every 30 secs.

Running the application 🡪

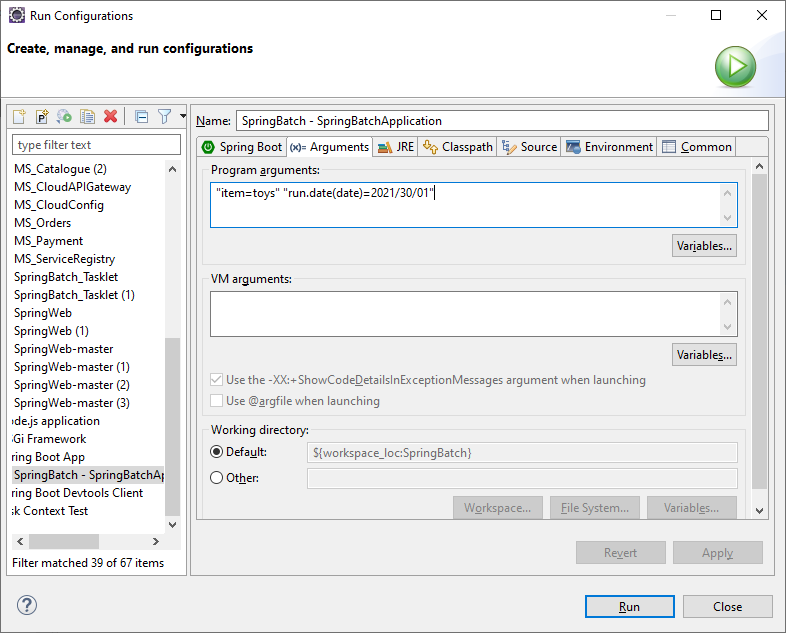
To provide job parameters in eclipse –

Right click -> run as -> run configuration -> click on arguments

Provide program argument -

"item=toys" "run.date(date)=2021/30/01"

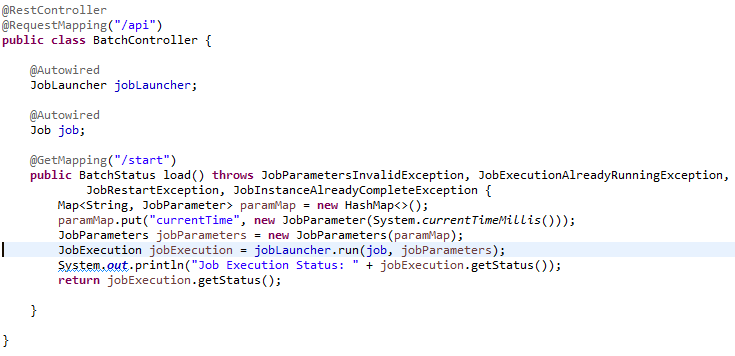
Change the argument while running the application, else the execution will fail for same job parameters.



Job parameters can also be configured programmatically. Refer scheduler ex.

**To start the job using rest API –**

1. Add spring.batch.job.enabled=false in application.properties . This will disable the job from starting immediately on launch.
2. Configure job parameters and invoke the url –



<http://localhost:8082/api/start>

To provide job parameters while running jar, execute below command –

java -jar SpringBatch-0.0.1-SNAPSHOT.jar "item=toys" "run.date(date)=2020/01/01"