Batch Processing

1. Bulk Processing
2. Banking: Bulk money transfer at night.
3. E-Commerce/Retail: Uploading items in bulk at night.

Spring Batch

1. It is a lightweight batch framework based on spring.
2. It is used for high volume batch processing.
3. Various I/O options like JSON, XML, SQL databases.
4. Easy to manage with start/stop options.
5. Retry and skip mechanism.

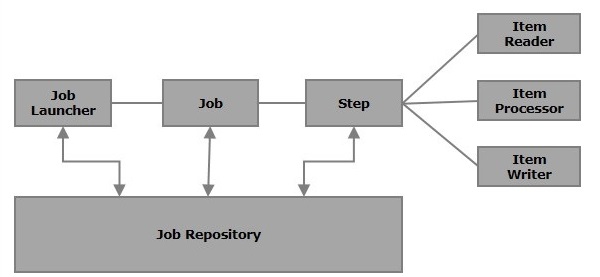
What is job and step?

* Job can be online transaction and step is like multiple steps in job like deduct money from the sender then add money to receiver.

Step types in Spring Batch

1. **Tasklet Step**: Online bulk transaction at night.
2. **Chunk-Oriented Step**: Processing chunk of CSV at a time. Like 5 rows at a time. The flow here is Item Reader -> Item Processor -> Item Writter

**Spring Batch Architecture**



We must add Spring Batch dependency in pom.xml

We must add @EnableBatchProcessing in main class

We must have at least one sql database connection otherwise spring batch application will not run.

We can use command line arguments or parameters to uniquely run one job

One can use RunIdIncrementer class but for that for the first time we have to pass some unique job parameters

ItemProcessor is not a mandatory step.

We can have chunk-oriented step and tasklet step in the same job

By default, spring batch will run all the jobs on the boot-up which have @Bean annotation. For local running it is helpful but for not for production, so we must schedule our spring batch job and we must run job manually on demand or in a scheduled way.

To prevent auto start we can add this config to property file **spring.batch.job.enabled: false**

We can either start via rest call or via spring schedular

When we manually stop any job then that job will be stop but the chunk whatever it was working that will be completed then gracefully it will be stopped.

Item readers can be anything:

1. Flat file (CSV)
2. JSON
3. XML
4. Database
5. Rest Api

For each of these source spring batch provides different methods

Flat file item reader:

1. Source location of CSV
2. Line Mapper:
3. Line Tokenizer (,)
4. Bean Mapper (converting from one line to object)

Flat file item reader CSV

If we change the order of the name here: **delimitedLineTokenizer.setNames("id", "first\_name", "last\_name", "email");**

then the value of the fields of the POJO class will also be changed.

If we remove any name from here, we will get parse exception.

We can change the delimiter from comma to pipe separator.

We can also provide the csv path from job arguments; we can hold that from **@Value(“#{jobParameters[‘inputFile’]}”) FileSystemResource fileSystemResource**

If we mark it **flatFileItemReader()** as a **@Bean** then we must add **@StepScope**

Also, we can use the **classPathResource** if the file is inside of **“src/main/resources”**

When we are reading from JSON then our model class attribute and json object attributes should be same otherwise it will create an exception. If we have any mismatch, then we have to use **@JsonProperty** with the name with the json file. Like the following:

@JsonProperty("first\_name")

**private** String firstName;

If there is unknown property in json file and there is no mapping for that in POJO class, then it will give an exception. To remove this exception, we must not read all the properties from JSON. So we can add this: **@JsonIgnoreProperties(ignoreUnknown = true)**

**jsonItemReader.setMaxItemCount(8):** Maximum read count is 8

**jsonItemReader.setCurrentItemCount(2):** Skip 2 documents from the top

XML reader

When we are reading xml file then we have to give the xml root element to POJO class. We have to annotate the pojo class with this: **@XmlRootElement(name="students")**

If there is any mismatch in property name, then we can use @XmlElement(name="firstName")

Converting java object to XML called as **Marshalling** and converting XML to java called as **Unmarshalling.**

JDBC item reader:

If there is any mismatch in database table column name and POJO class attribute name use can use alias “as” in sql query.

**jdbcItemReader.setMaxItemCount(8):** Maximum read count is 8

**jdbcItemReader.setCurrentItemCount(2):** Skip 2 documents from the top

Item reader is only capable of reading one item at a time so we to think of a design to get one element at a time. There may be a case where we are calling an api which is returning us list of users but in item ream we are only taking care of one item

Item writers can be anything:

1. Flat file (CSV)
2. JSON
3. XML
4. Database
5. Rest Api

For any pre-processing of data before writing we can do that in ItemProcessor

**JsonWriter**

If any time we want to change the property name while writing in json then in the respective attribute in POJO we can add **@JsonProperty(“first\_name”)**

Fault Tolerance:

1. Skip Policy
2. Retry Mechanism

With no fault tolerance, if there is any bad record then spring batch will throw exception and from that record the spring batch program will stop

We can use faultTolerant() with skip and skipLimit

skip(FlatFileParseException.class) means it will skip if there is any FlatFileParseException

skipLimit(1) mean it will skip 1 bad record, but if we have any other bad record then again there will be exception, so then its better to pass Integer.MAX\_VALUE or we can use skipPolicy with AlwaysSkipItemSkipPolicy object.

We can have more than one skip method to handle multiple type of exception. It isalmost like a try catch.

But we should know which item cause the issue on itemreader, itemprocessor, itemwritter.

So, we can have a skiplistner to intercept this

We can create skip listener with annotation or implementing SkipListener interface

Retry mechanism is only applicable for item processor and item writer

We can use this with retryLimit(1)

Retry mechanism will not work with skipLimit(Integer.MAX\_VALUE). It will fall into infinite loop

Spring batch will retry for processor with n times and writer with n-1 times

**Suppose we have a project of migrating all student data from postgres to mysql**

1. We will create 2 entity class corresponding to the postgres student table schema and mysql student table schema.
2. We will create 2 datasource. One for postgres and one for mysql.
3. With both datasources we will create 2 entityManagerFactory as we will be using JPA rather than hibernate.
4. We will create one JpaItemReader with the postgres entity manager and create JpaItemWriter with mysql entity manager.
5. We need one transaction manager of mysql as we will be writing to mysql database.
6. We can have fault tolerance, skip policy, retry mechanism, skip listener and other job and step listener we per our need.
7. We can use currentItemCount and maxItemCount to set the range, if we don’t want all the task to be happening in the one job.
8. We can configure these currentItemCount and maxItemCount from the command line.
9. We can use rest call to run the job manually with different job parameters or we can use spring scheduler to run the job every night with different parmeter.