Spring Batch

Lesson 02 : Examining Jobs and Steps

Lesson Objectives

- Chunk vs. Item Processing
- Jobs & Steps
- ItemReader interface
- ItemProcessor interface
- ItemWriter interface
- Job Execution
- Examples

Chunk vs. Item Processing

- **Spring batch provides two options while deciding what a unit of data to be processed is:**
  - **an individual item** : consists of a single object that typically represents a single row in a database or file.
    - Item-based processing, therefore, is the reading, processing, and then writing of your data one row, record, or object at a time
  - **a chunk of items** : is a subset of the records or rows that need to be processed, typically defined by the commit interval.
    - In Spring Batch, chunk is defined by how many rows are processed between each commit.
    - although each row is still read and processed individually, all the writing for a single chunk occurs at once when it's time to be committed.
    - Chunks are defined by their commit intervals. If the commit interval is set to 50 items, then your job reads in 50 items, processes 50 items, and then writes out 50 items at once.
    - Because of this, the transaction manager plays a key part in the configuration of a chunk-based step.

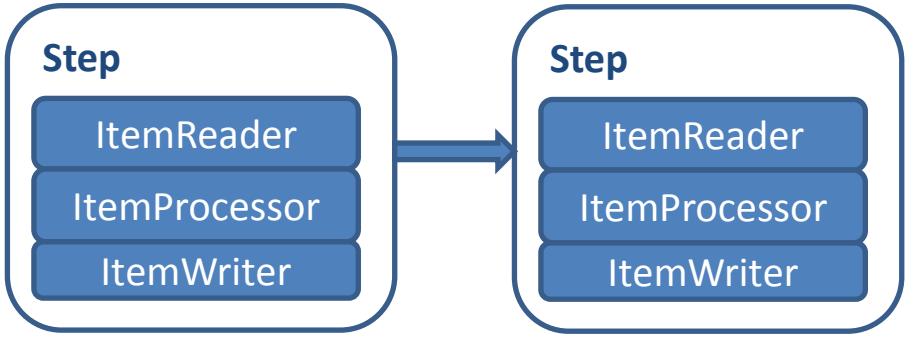
Chunk Processing : example

```
<job id="stepJob">
    <step id="step1">
        <tasklet>
            <chunk reader="inputReader" writer="outputWriter"
                    commit-interval="50"/>
        </tasklet>
    </step>
</job>
```

- **In the example, you're defining what a chunk is for your step.**
- **You're saying to use the inputReader bean (an implementation of the ItemReader interface) as the reader and the outputWriter bean (an implementation of the ItemWriter interface) as the writer, and that a chunk consists of 50 items.**

“Chunk Oriented” implementation: Jobs & Steps

- **A batch job is a collection of steps in a specific order to be executed as part of a predefined process.**
  - Normally configured via XML.
- **Step is a self-contained unit of work that is the main building block of a job.**
  - Each step has up to three parts: an ItemReader, an itemProcessor, and an ItemWriter.
  - ItemReader : a strategy interface that provides the ability to input items.
  - ItemProcessor : a facility to apply business logic to an individual item as provided.
  - ItemWriter : a strategy interface that provides the ability to output a list of items.



ItemReader interface

- **ItemReader is responsible of reading input data.**
  - Its main method is Object read() which returns the next item each time it is invoked, and null when the data is exhausted.
  - This returned item is usually expected to map to a domain object.
- **The resulting output of the ItemReader is collected into a list that is used to apply the business rules.**
  - There are many default implementations of ItemReader like **AbstractCursorItemReader** which reads a record from the database, FlatFileItemReader which reads contents from a file, JmsItemReader which reads the message from a JMS destination etc
- **Each reader typically contains the following properties:**
  - resource - the location of the file to be imported
  - lineMapper - the mapper to be used for mapping each line of record
  - lineTokenizer - the type of tokenizer
  - fieldSetMapper - the mapper to be used for mapping each resulting token

ItemProcessor interface

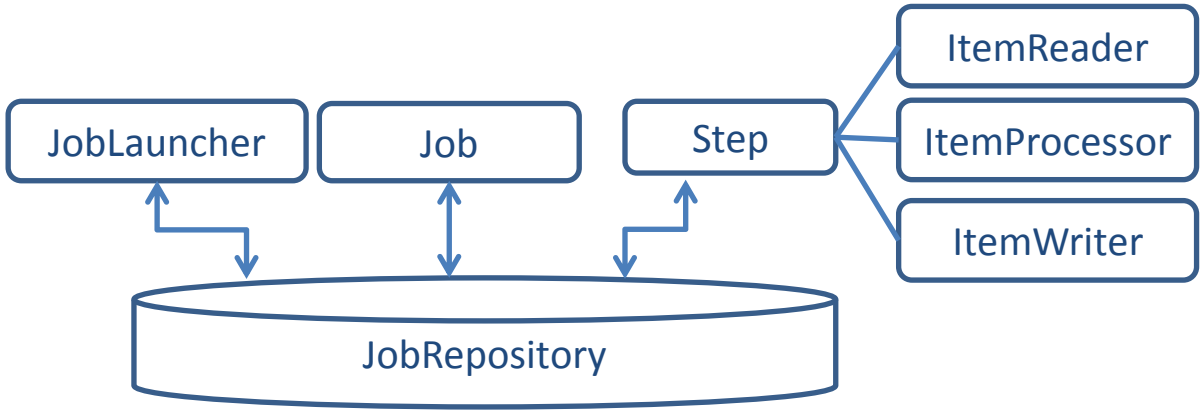
- **The ItemProcessor interface has one "process" method that is used for item transformation through applied business rules.**
  - Given an input item, which is one item resulting from the output of the ItemReader, apply the business rules and the processor either returns the modified item or a new item for continued processing.
  - Or if continued processing of the item should not take place, the ItemProcessor should return a null value effectively filtering out the item.
  - You also have the ability to chain processors together to apply very complex business rules, with the output of one processor becoming the input of the next processor in the chain and so on.
- **Within the ItemProcessor implementation is where the bulk of the work by the developer will be as this is where most of your business logic will be applied.**
  - The resulting output of the ItemProcessor is collected into a list that will then be fed to the ItemWriter for output processing.

ItemWriter interface

- **The ItemWriter interface has one “write” method and that is called one time for the chunk being processed and is supplied the list items for generic output.**
- **There are many default implementations of ItemWriter that have been provided by Spring Batch such as FlatFileItemWriter, JdbcBatchItemWriter, and JpaItemWriter to name a few.**

Job Execution

- **Most of the components share a JobRepository, which persists information about the job and step executions**
- A step goes through a list of items as read in by the ItemReader.
- As the step processes each chunk of items, the StepExecution in the repository is updated with where it is in the step.



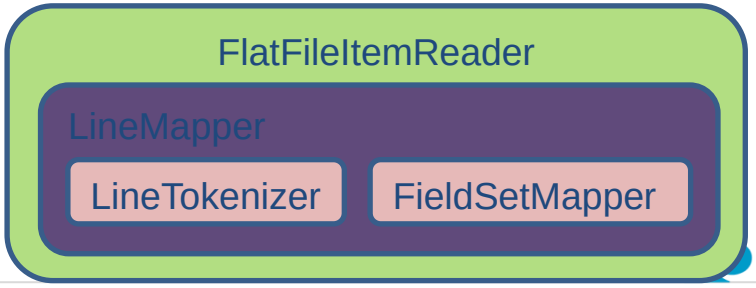
Example follows……

- **This reads employee details from a comma-separated text file, assigns a rank based on salary and stores results back into another text file**

Reading from a flat file : FlatFileItemReader

- **The FlatFileItemReader consists of two main components:**
  - a Spring Resource that represents the file to be read and an implementation of the LineMapper interface.
  - The LineMapper serves a similar function as the RowMapper does in Spring JDBC.
  - This class will take each line of a file as a String and convert it into a domain object (item) to be processed.

```
<bean name="empReader"
class="org.springframework.batch.item.file.FlatFileItemReader">
  <property name="lineMapper" ref="employeeLineMapper" />
  <property name="resource" value="inputdata.txt" />
</bean>
```



LineMapper interface

- **With LineMapper, a provided String represents a single record from a file.**
  - There is a two-step process for getting this string to the domain object you will later work with. These two steps are handled by the LineTokenizer and FieldSetMapper.
  - A LineTokenizer implementation parses the line into a FieldSet. In order to be able to map the individual fields of each record to your domain object, you need to parse the line into a collection of fields.
  - The FieldSet in Spring Batch represents that collection of fields for a single row.
  - The FieldSetMapper implementation maps the FieldSet to a domain object.

```
<bean name="employeeLineMapper"
class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
   <property name="lineTokenizer" ref="defaultTokenizer" />
   <property name="fieldSetMapper" ref="employeeFieldSetMapper" />
</bean>
```

LineMapper : Working with fixed length records

- **Example of fixed-length file.**

```
Michael  TMinella    123  4th Street      Chicago     IL60606
Warren   QGates      11   Wall Street     New York    NY10005
Ann      BDarrow     350  Fifth Avenue    New York    NY10118
Terrence HDonnelly   4059 Mt. Lee Drive Hollywood
CA90068
```

```
<bean class="org.springframework……DefaultLineMapper">
    <property name="lineTokenizer">
        <bean class= "org.springframework……FixedLengthTokenizer">
           <property name="names" value="fnm, lnm, st, addr, city, code" />
            <property name="columns" value="1-32,33-40,41-55,56-57,58-58" />
        </bean>
    </property>
…..
```

FieldSetMapper interface

- **A FieldSetMapper transforms the line (split by the LineTokenizer) into a domain object – you'll need to write your own implementation for this**

```
public class EmpFieldSetMapper implements FieldSetMapper<Emp> {
   public Emp mapFieldSet(FieldSet fieldSet) throws BindException {
      if(fieldSet == null) return null;
      Emp emp = new Emp();
      emp.setEmpid(fieldSet.readInt(0)); // index is 0 based
      emp.setEname(fieldSet.readString(1));
      …
      return emp;
   }}
```

  - The FieldSet parameter from the LineTokenizer is a flat file equivalent to the JDBC ResultSet: it helps retrieve field values and do some conversion between String and richer object like Date.

Example….

- **Assume that we have a text file that contains comma-separated values**

```
7876,ADAMS,CLERK,1100
7499,ALLEN,SALESMAN,1600
7698,BLAKE,MANAGER,2850
….
```

- **A simple bean that represents a single Employee**

```
public class Employee {
   Integer empid, salary;
   String ename, title, rank;
    // set/get methods for all properties
}
```

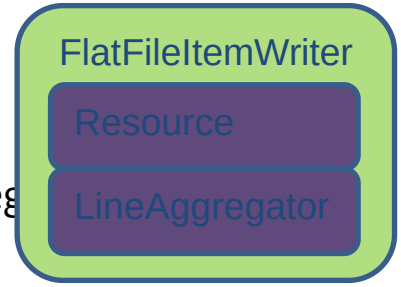
ItemProcessor

- **Spring Batch lets you do processing on reader output.**
  - This processing can do virtually anything to the output before it gets passed to the writer, including changing the type of the data.
  - In our example, we are assigning a Rank based on the salary amount. The item processor takes an input Bean and converts it to an output bean. In this case the beans are the same but they don't have to be.

```
public class EmpProcessor implements ItemProcessor<Emp, Emp> {
   public Emp process(Emp emp) throws Exception {
      if(emp.getSalary() >= 2500 )
         emp.setRank("Director"); // if salary >= 2500, set rank as "Director"
      else
            emp.setRank("N/A");
      return emp;
   }
}
```

FlatFileItemWriter: ItemWriter implementation

- **FlatFileItemWriter is used to generate text file output.**
  - This class addresses the issues with file-based output in Java with a clean, consistent interface for you to use.
  - FlatFileItemWriter consists of a resource to write to and a LineAggreg implementation.
  - LineAggregator is used to convert an individual item to a String for output




```
<bean id="empWriter" class="org.springframework… FlatFileItemWriter">
   <property name="resource" value="file:target/outputdata.txt" />
   <property name="lineAggregator">
     <bean class="org.….DelimitedLineAggregator">
        <property name="delimiter" value="," />
        <property name="fieldExtractor">
           <bean class="org.…BeanWrapperFieldExtractor">
             <property name="names" value="empid,ename,salary,rank" />
           </bean></property></bean></property> </bean>
```