Batch

TOC

Overview

Why do we need Batch Jobs?

- Batch applications process large amounts of data without human intervention.
- This processing is often executed on a regular basis (for example, daily, weekly, or monthly)
- May be long running.
- Data usually can’t fit into memory or a single transaction
- **For example :** To compute data for
- Generating monthly financial statements,
- Indexing files
- Sending subscription e-mails
- Sending monthly invoices
- Synchronizing a data warehouse
- Performing business reporting
- Processing orders

Batch Jobs

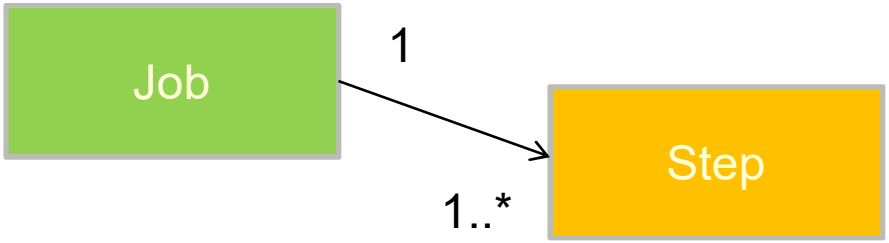
- The most common scenario for a batch application is exporting data to files from one system and processing them in another.

- Imagine you want to exchange data between two systems: you export data as files from system A and then import the data into a database on system B.

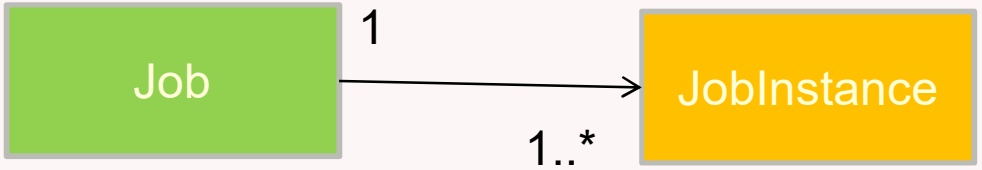
Spring Batch Introduction

Spring Batch Domain

- Job is constructed of one or more steps.
- Step represents this partial processing of work performed by Job.

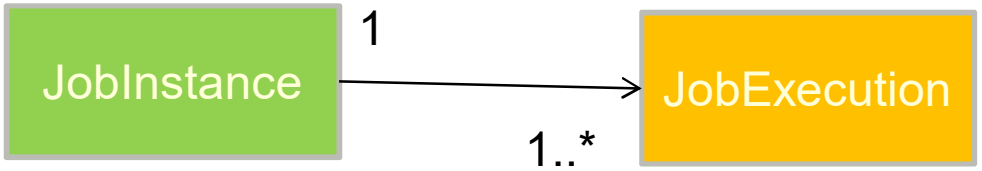


- Every time the Job is executed, a new JobInstance is created so one Job can have one or more JobInstances
- A Job can be executed daily , monthly etc.
- Each time , we have a JobInstance example : Monday Jan 2016 JobInstance , Tue Jan 2106 JobInstance etc



Spring Batch Domain

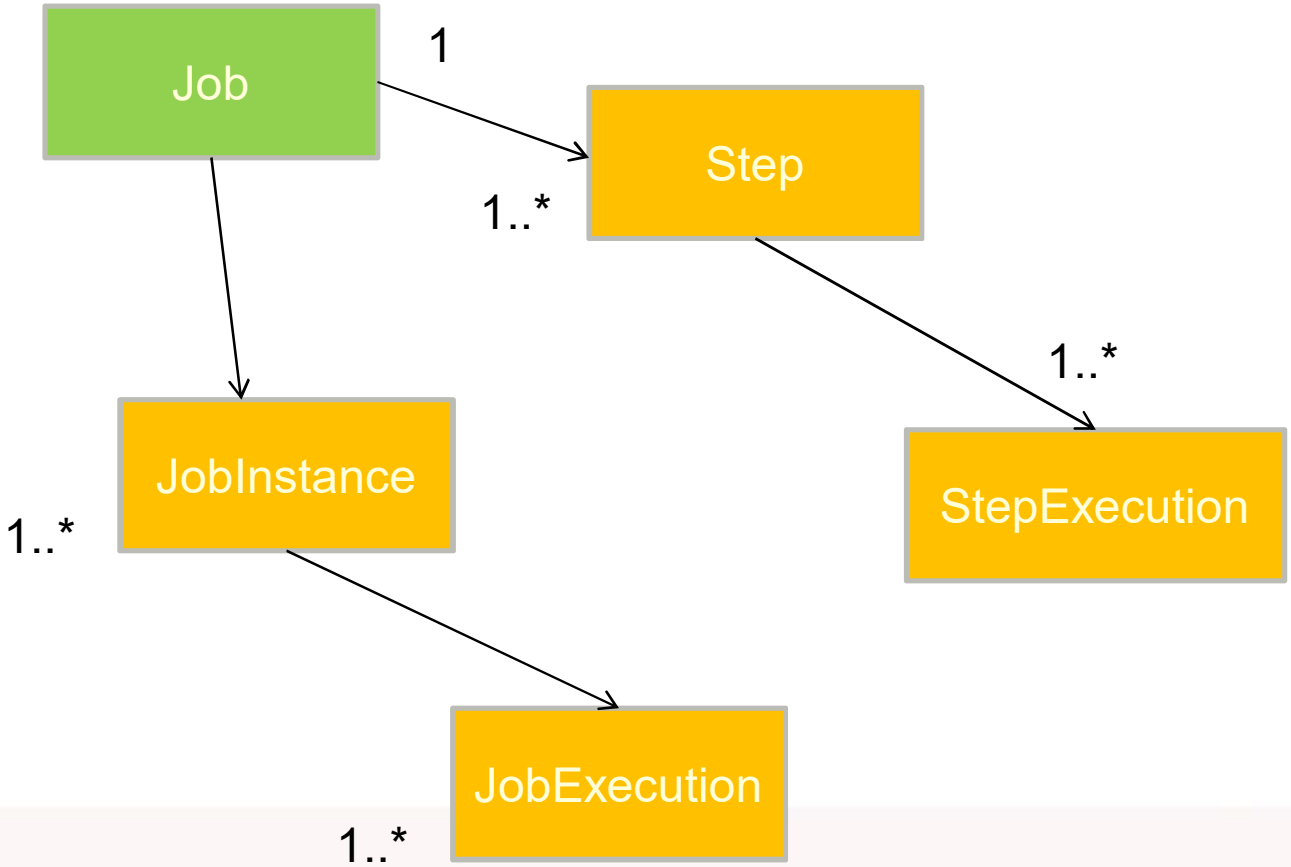
- One job instance can be executed various times (for example, when JobInstance needs to be restarted because of failure),so it can have various **JobExecutions**



- In the same way , a step has many StepExecution

- Each Job needs to have at least one Step, it create a StepExecution for each JobExecution.

Spring Batch Domain – Complete Diagram

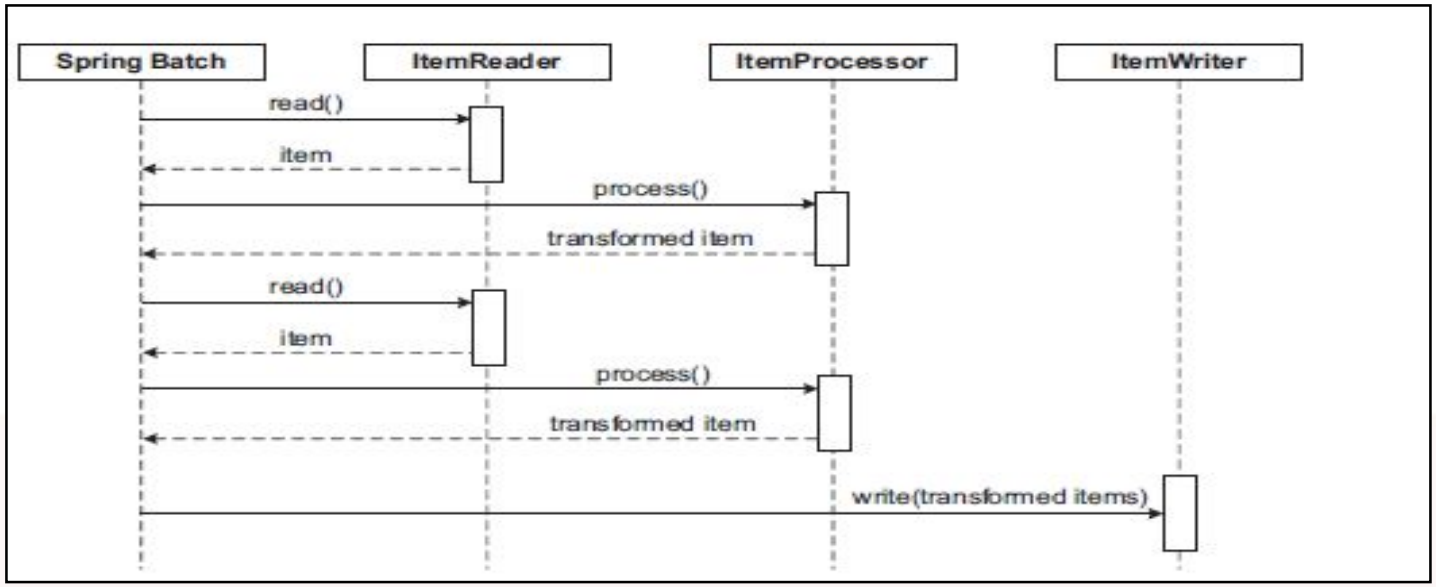


Steps

- **Steps can be written in two ways :**
- Chunk oriented Steps
- Tasklet

Chunk Oriented Processing

- Step is composed of one mandatory ItemReader<T>, one **optional** ItemProcessor<T, S>, and one mandatory ItemWriter<S>.
- ItemReader reads the data one at a time and ItemProcessor transforms/filters the data read by ItemReader
- Spring Batch collects items one at a time from the item reader & Processor into a configurable-sized chunk and send to Item Writer
- Chunk processing is particularly well suited to handle large data operations because a job handles items in small chunks instead of processing them all at once

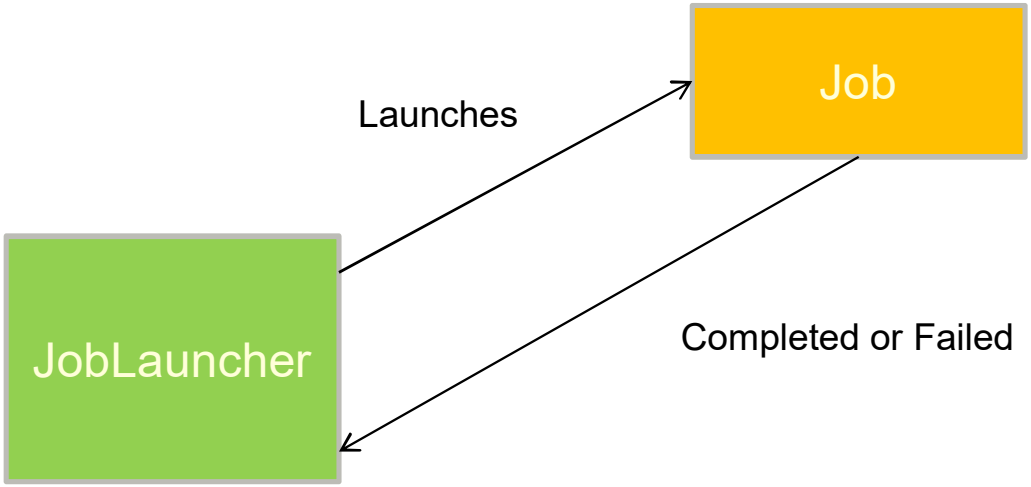


- The loop of reading multiple chunks ends when ItemReader returns null, which means that there's no more data to process

Tasklet Step

- Chunk-oriented processing is not the only type of step we need to cover for enterprise application use cases.
- Sometimes we need to perform a single action as part of a bigger flow.
- For example, we might need to send a notification at the end of a job or perform a single stored procedure call.
- Spring Batch provides the Tasklet interface.
- It has only one method, execute, where we can place our custom logic

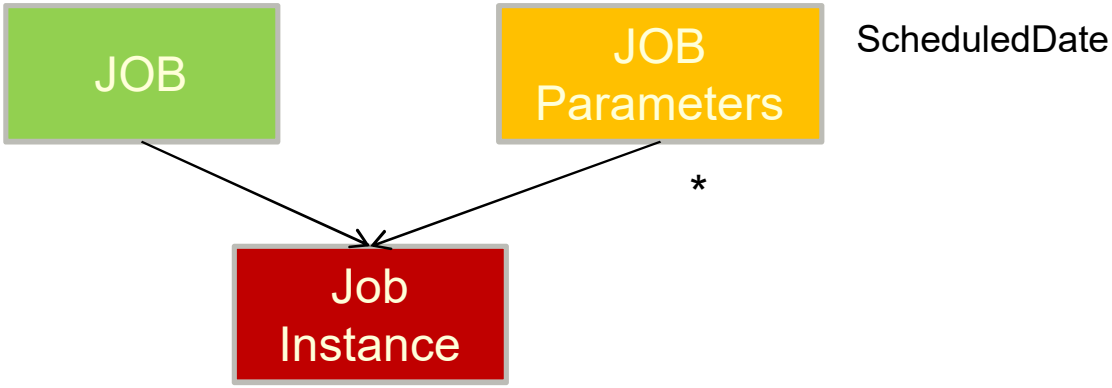
Launching Job



- JobLauncher declares only one method, run(), which returns an instance of JobExecution
- Both Synchronous and Asynchronous calls are possible, will see that later
- Using JobExecution.getExitStatus(). , we can get the status of Batch job

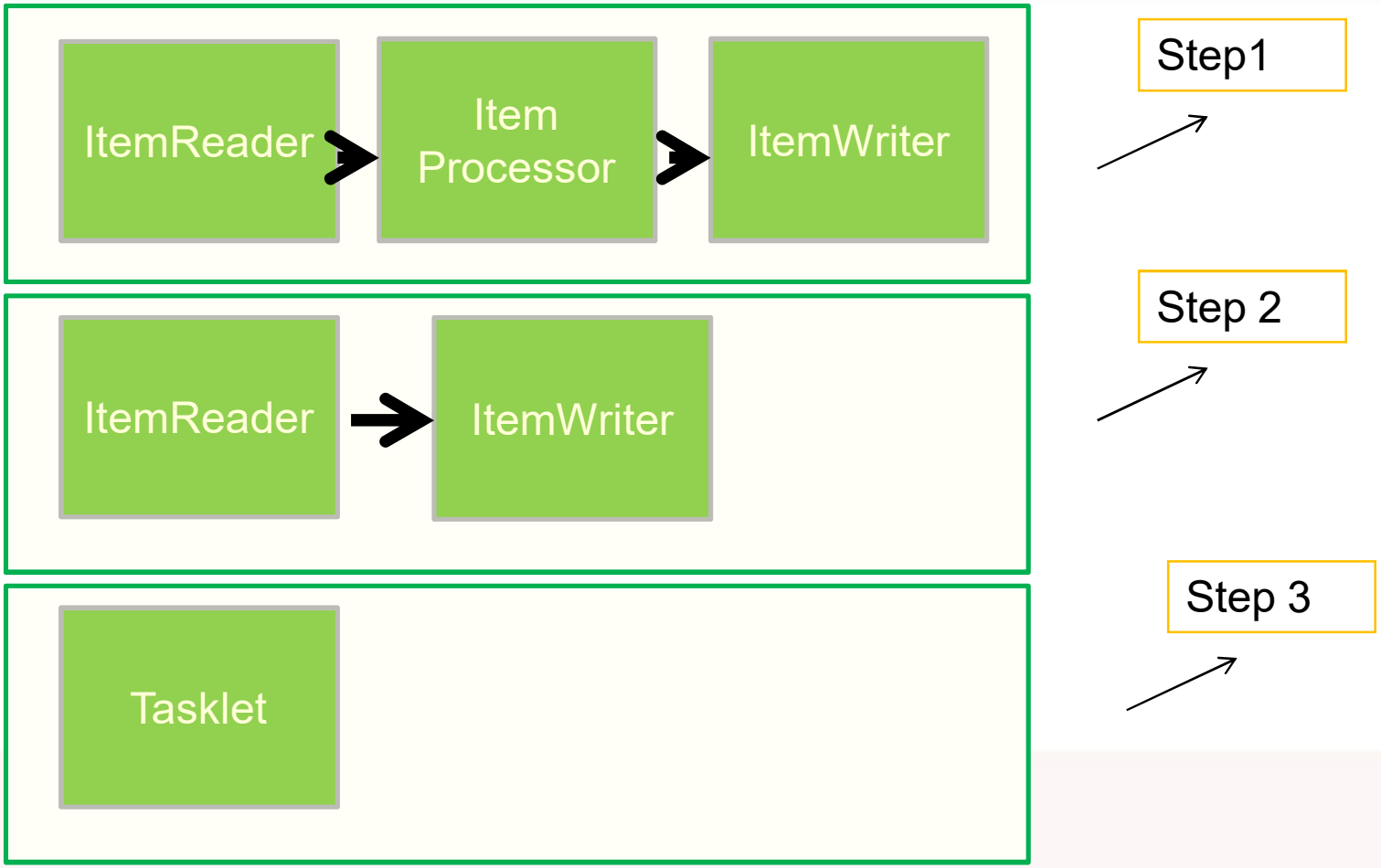
Job Identity and Job Parameters

- Need to pass parameters into a batch Job.
- Provides this support via the JobParameters class, which can be passed into the JobLauncher.run() method.
- This class encapsulates parameters into Map<String, JobParameter>.
- So each parameter has its name (key), and the value can be any Java type.
- JobParameters (plural) represents all parameters for one JobInstance.



- Job parameters are job specific. Most of the time, you'll be using a timestamp or a sequence to change the job identity for each run.
- JobInstanceAlreadyCompletedException
- Unique Parameter such as Date / Time

Example Job



ItemReaders / ItemWriters

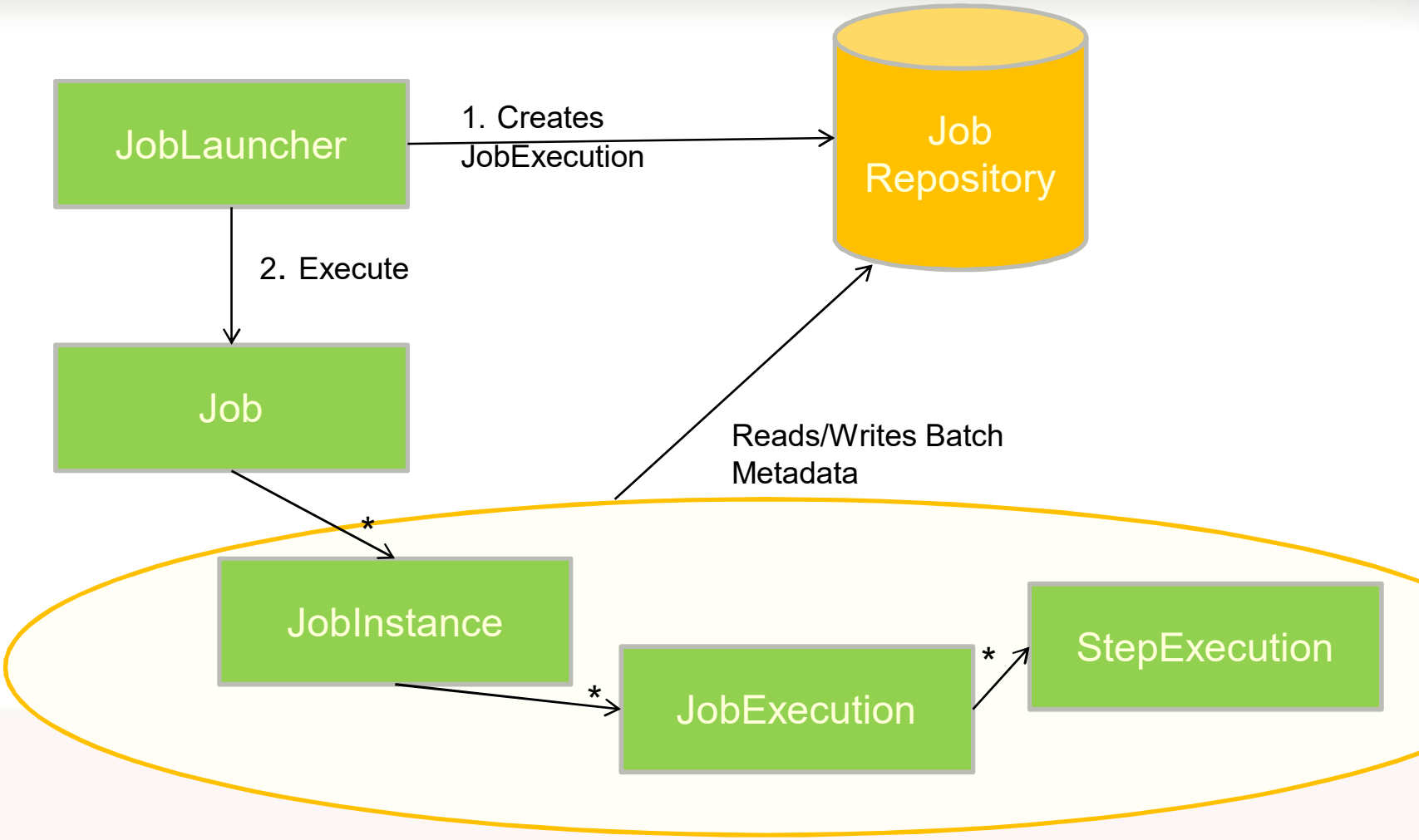
- Spring Batch provides some commonly used implementation out of the box for reading/writing from/to the JDBC/Hibernate/stored procedure , for example :-

- Cursor-based item readers—the cursor is a DB construct in which rows are streamed from a database
- Paging-based item readers—uses a distinct where clause for each chunk of data (page)
- Flat files
- XML
- JMS
- For simple jobs , we can use off the shelf components
- Otherwise we can write our custom components

DEMO

01-BatchBasic
02-BatchBasicTwoSteps
02-BatchBasicTwoSteps-JavaConfig

Job Data Persistence to Job Repository



Job Data Persistence to Job Repository

- Spring Batch uses a defined database schema to store its metadata and provides SQL scripts to create or drop the schema for the most commonly used relational databases.
- These schemas are located in the spring-batch-core library in the org/ springframework /batch/core folder.
- So, for example, if we want to create a schema for the MySQL database, we can use the SQL script classpath:org/springframework/batch/core/schema-mysql.sql to create the SB schema
- And the script classpath:org/springframework/batch/core/schema-drop-mysql.sql to erase it.

```
<jdbc:initialize-database data-source="dataSource">
    <jdbc:script location="dbscripts.sql"/>
    <jdbc:script location="classpath:org/springframework/batch/core/schema-drop-mysql.sql"/>
    <jdbc:script location="classpath:org/springframework/batch/core/schema-mysql.sql"/>
</jdbc:initialize-database>
```

Job Data Persistence to Job Repository

```
<batch:job-repository id="jobRepository"/>
```

- This is equivalent to :-

```
<batch:job-repository id="jobRepository" data-source="dataSource" transaction-manager="transactionManager"/>
```

SCHEMA OVERVIEW

BATCH_JOB_INSTANCE

JOB_INSTANCE_	VERSION	JOB_NAME	JOB_KEY
1	0	UnZipAndreadWriteJob	879ccde03cad29b90cc11f683b60f34e

One instance with “UnzipAndReadWriteJob” JOB_NAME

BATCH_JOB_EXECUTION

JOB_EXECUTION_	VERSION	JOB_INSTANCE_ID	CREATE_TIME	START_TIME	END_TIME	STATUS	EXIT_CODE	EXIT_MESSAGE	LAST_UPDATED
1	2	1	2016-09-14	2016-09-14 23:	2016-09-14 23	FAILED	FAILED	org.springframework.batch.item.file.FlatFileParseExcep	2016-09-14 23:5
2	2	1	2016-09-15	2016-09-15 00:	2016-09-15 00	FAILED	FAILED	org.springframework.batch.item.file.FlatFileParseExcep	2016-09-15 00:0
3	2	1	2016-09-15	2016-09-15 00:	2016-09-15 00	FAILED	FAILED	org.springframework.batch.item.file.FlatFileParseExcep	2016-09-15 00:0
4	2	1	2016-09-15	2016-09-15 00:	2016-09-15 00	FAILED	FAILED	org.springframework.batch.item.file.FlatFileParseExcep	2016-09-15 00:0
5	2	1	2016-09-15	2016-09-15 00:	2016-09-15 00	FAILED	FAILED	org.springframework.batch.item.file.FlatFileParseExcep	2016-09-15 00:0
6	2	1	2016-09-15	2016-09-15 00:	2016-09-15 00	FAILED	FAILED	org.springframework.batch.core.StartLimitExceededExcep	2016-09-15 00:0
7	2	1	2016-09-15	2016-09-15 00:	2016-09-15 00	FAILED	FAILED	org.springframework.batch.item.file.FlatFileParseExcep	2016-09-15 00:0
8	2	1	2016-09-15	2016-09-15 00:	2016-09-15 00	FAILED	FAILED	org.springframework.batch.item.file.FlatFileParseExcep	2016-09-15 00:0

- For JOB_INSTANCE_ID “1”, how many executions we have(so many executions because of failure)
- JOB_EXECUTION_ID : 1,2,3 till 8

SCHEMA OVERVIEW

BATCH_JOB_EXECUTION_PARAMS

JOB_EXECUTION_	TYPE_CD	KEY_NAME	STRING_VAL	DATE_VAL	LONG_VAL	DOUBLE_VAL	IDENTIFYING
1	LONG	JobID		1970-01-01 05:30:00	1	0	Y
1	STRING	inputResource	classpath:ZipFile/products.zip	1970-01-01 05:30:00	0	0	Y
1	STRING	targetDirectory	./target/ZipFileOutput	1970-01-01 05:30:00	0	0	Y
1	STRING	targetFile	products.txt	1970-01-01 05:30:00	0	0	Y

Parameters passed to the job

BATCH_STEP_EXECUTION

STEP_EXECUTION_	VERSION	STEP_NAME	JOB_EXECUTION_ID	START_TIME	END_TIME	STATUS	COMMIT_COUNT	READ_COUNT	FILTER_COUNT	WRITE_COUNT	READ
1	3	unZipStep	1	2016-09-14 23:57:24	2016-09-14 23:57:24	COMPLETED	1	0	0	0	
2	2	readerWriter	1	2016-09-14 23:57:24	2016-09-14 23:57:24	FAILED	0	1	0	0	

- Step details , Job_execution_id is the foreign key

Launching Job

Configuring JobLauncher

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

<batch:job-repository id="jobRepository" />
```

Launching Job - 1

- **Look up the JobLauncher and Job**
- Launch it from anywhere

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"classpath:01batch-configWithoutScheduler.xml"})
public class BatchTest {

    @Autowired
    private JobLauncher jobLauncher;

    @Autowired
    private Job job;


    @Test
    public void testReadAndWrite() throws JobExecutionAlreadyRunningException, JobRestartExcept
    {
        JobParameters jobParameters= new JobParametersBuilder()
                .addLong("JobID", 1L)
                .addString("inputResource", "classpath:ZipFile/products.zip")
                .addString("targetDirectory", "./target/ZipFileOutput")
                .addString("targetFile","products.txt")
                .toJobParameters();


        jobLauncher.run(job,jobParameters);
    }
```

- Using this in all examples

Launching Job - 2

- **Use Scheduler**

```
<!-- Run every 5 seconds -->
    <task:scheduled-tasks>
        <task:scheduled ref="runScheduler" method="run" cron="*/5 * * * * MON-FRI"/>
    </task:scheduled-tasks>

 <bean id="runScheduler" class="com.way2learnonline.batch.RunScheduler"/>
```

- Java Config :-

```
@Scheduled(cron="*/5 * * * * MON-FRI")
public void scheduleJob()
{
    runScheduler().run();
}


@Bean
public RunScheduler runScheduler() {
    return new RunScheduler();
}
```

DEMO

Readers

ItemReader

```
public interface ItemReader<T> {
  T read() throws Exception, UnexpectedInputException,
                  ParseException, NonTransientResourceException;
}
```

- ParseException - if there is a problem parsing the current record
- NonTransientResourceException - if there is a fatal exception in the underlying resource.
- UnexpectedInputException - if there is an uncategorised problem with the input data.

- ItemReader interface that provides a contract for reading data.

- This interface supports generics and contains a read method that returns the next element read.

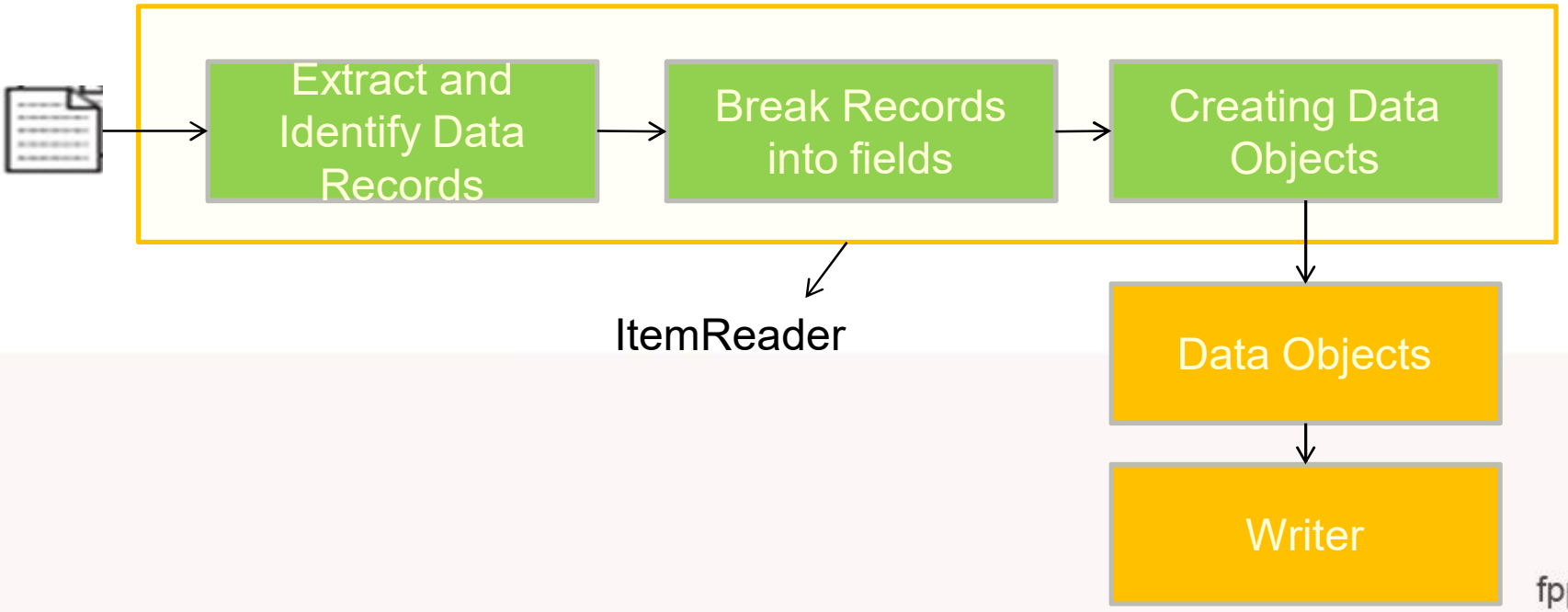
ItemReader

- Spring Batch has many ItemReader implementations available :-
- ItemReader API
- We will discuss few File and Database related Item Readers

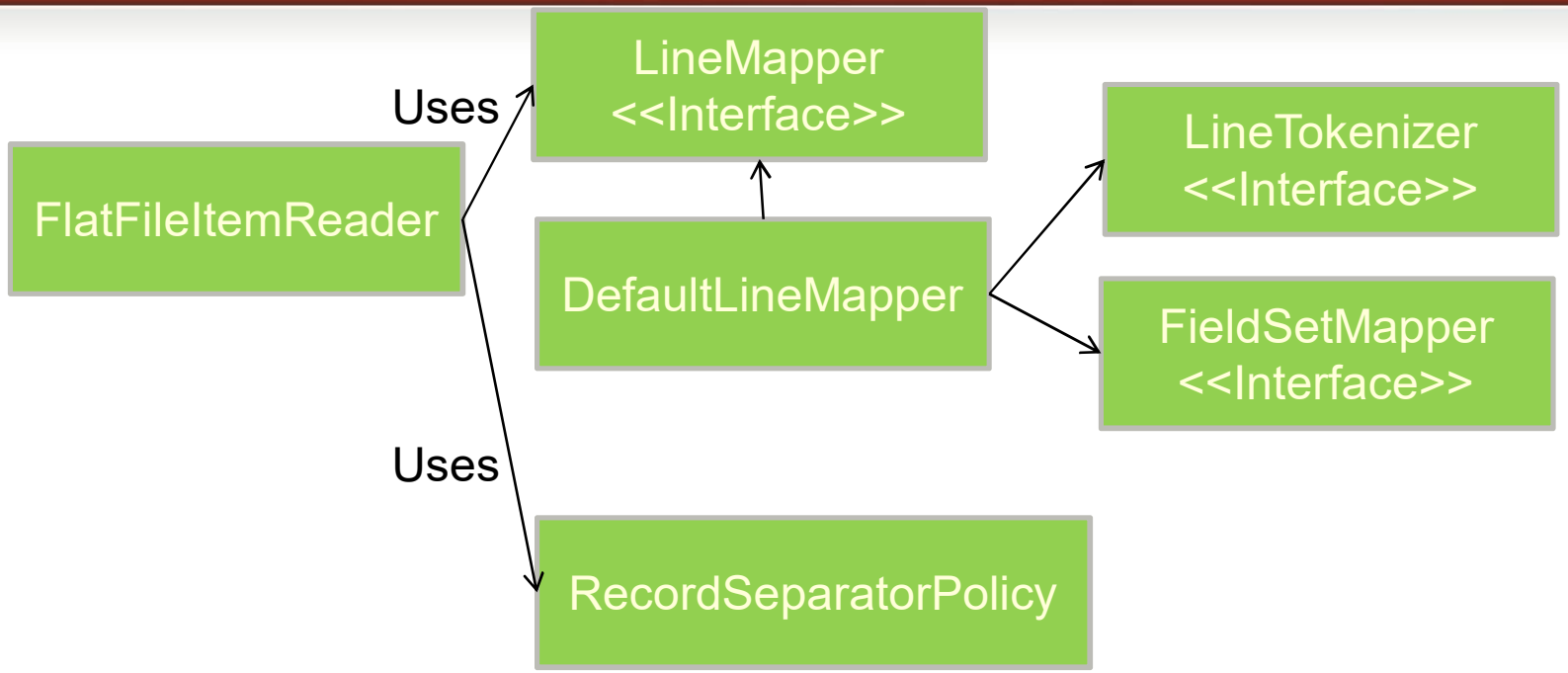
Reading Flat Files

- Data Files
- May contain Header Line
- Comma separated or other separator
- Or fixed length

- ItemReader used : **FlatFileItemReader**



Reading Flat Files



LineMapper
Maps a data line to a data object
LineTokenizer
Splits a data line into tokens; invoked by the **DefaultLineMapper** class, the default implementation of the LineMapper interface
FieldSetMapper
Creates a data object from tokens; invoked by the **DefaultLineMapper** class, the default implementation of the LineMapper interface
RecordSeparatorPolicy
Identifies beginning and end of data records

Reading comma Separated File

```
<!-- Reader defined -->
<bean id="productFileReader" class="org.springframework.batch.item.file.FlatFileItemReader">
    <property name="resource" value="classpath:Files/products.txt"/>
    <property name="linesToSkip" value="1"/>
    <property name="lineMapper">
        <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
            <property name="lineTokenizer">
                <bean class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
                    <property name="names" value="ID,NAME,DESCRIPTION,PRICE"/>
                </bean>
            </property>
            <property name="fieldSetMapper">
                <bean class="com.way2learnonline.batch.support.ProductFieldSetMapper"/>
            </property>
        </bean>

    </property>
</bean>
```

Tokenized Into Fields

Name of Fields

Custom Implementation - FieldSetMappe uses these fieldNames and create Domain object

```
PRODUCT_ID,NAME,DESCRIPTION,PRICE
PR....210,BlackBerry 8100 Pearl,desc1,124.60
PR....211,Sony Ericsson W810i,desc2,139.45
PR....212,Samsung MM-A900M Ace,desc3,97.80
PR....213,Toshiba M285-E 14,desc4,166.20
```

- You have already seen a demo of this

Reading FixedLength File

```
<!-- Reader defined -->
<!-- org.springframework.batch.item.file.transform.FixedLengthTokenizer used in FlatFileItemReader -->
<bean id="productFileReader" class="org.springframework.batch.item.file.FlatFileItemReader">
    <property name="resource" value="classpath:Files/products.txt"/>
    <property name="lineMapper">
        <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
            <property name="lineTokenizer">
                <bean class="org.springframework.batch.item.file.transform.FixedLengthTokenizer">
                    <property name="columns" value="1-9,10-35,36-50,51-56"/>
                    <property name="names" value="ID,NAME,DESCRIPTION,PRICE"/>
                </bean>
            </property>
            <property name="fieldSetMapper">
                <bean class="com.way2learnonline.batch.support.ProductFieldSetMapper"/>
            </property>
        </bean>

    </property>
</bean>
```

Using FixedLengthTokenizer

DEMO

FieldSetMapper

- **ProductFieldSetMapper**
- Custom FieldSetMapper
- Implements FieldSetMapper interface
- **PassThroughFieldSetMapper**
- Provides the FieldSet without doing any mappings to objects.
- Useful if you need to work directly with the field set.
- **BeanWrapperFieldSetMappe**r
- FieldSetMapper that automatically maps fields by matching a field name with a setter on the object using the JavaBean specification.

```
<bean id="fieldSetMapper"
      class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper">
    <property name="prototypeBeanName" value="player" />
</bean>

<bean id="player"
      class="org.springframework.batch.sample.domain.Player"
      scope="prototype" />
```

- Fields don’t need to be value of file’s headers , field names in Domain class are in the order corresponding to the order of fields in Input File

Reading JSON File

```
<!-- Reader defined -->
<!-- JsonLineMapper in FlatFileItemReader -->
<bean id="productFileReader" class="org.springframework.batch.item.file.FlatFileItemReader">
    <property name="resource" value="classpath:Files/products.txt"/>
    <property name="lineMapper" ref="productLineMapper"/>
    <property name="recordSeparatorPolicy" ref="jsonRecordSeparatorPolicy"/>

</bean>

<bean id="productLineMapper" class="com.way2learnonline.batch.support.JsonLineMapperProductWrapper">
    <property name="delegate" ref="lineMapperType"/>
</bean>

<bean id="lineMapperType" class="org.springframework.batch.item.file.mapping.JsonLineMapper"/>

<bean id="jsonRecordSeparatorPolicy" class="org.springframework.batch.item.file.separator.JsonRecordSeparatorPolicy"/>
```

Written our own LineMapper which uses Spring Batch provided JsonLineMapper

DEMO

- Default JsonLineMapper return Map of objects and we need Product objects so a custom class is required

Notice , we have changed RecordSeparator policy

Reading XML File

```
<!-- Reader defined -->
<!-- Also check annotated Product class -->
<bean id="productReader" class="org.springframework.batch.item.xml.StaxEventItemReader">
    <property name="fragmentRootElementName" value="product"/>
    <property name="resource" value="classpath:Files/products.xml"/>
    <property name="unmarshaller" ref="jaxb"/>
</bean>

<bean id="jaxb" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
    <property name="classesToBeBound">
        <list>
            <value>com.way2learnonline.domain.Product</value>
        </list>
    </property>
</bean>
```

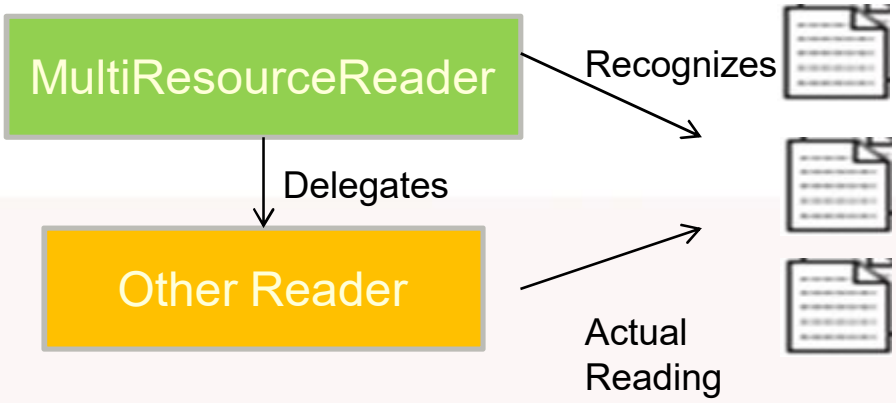
Notice StaxEventItemReader in place of FlatFileItemReader

- This Reader does not need LineMapper or FieldSet etc
- Check the documentation of Reader before using.
- StAX is a standard XML-processing API that streams XML data to your application.
- JAXB is used here , Other technologies can also be used like Castor,XMLBeans,JiBX,XStream

DEMO

Reading File Sets

- Input can enter an application as a set of files, not only as a single file or resource.
- For example, files can periodically arrive via FTP in a dedicated input directory.
- In this case, the application doesn't know in advance the exact filename, but the names will follow a pattern that you can express as a regular expression.



Reading File Sets

```
<bean id="multiResourceReader" class="org.springframework.batch.item.file.MultiResourceItemReader">
    <property name="resources" value="classpath:Files/products*.txt"/>
    <property name="delegate" ref="flatFileItemReader"/>
</bean>


<bean id="flatFileItemReader" class="org.springframework.batch.item.file.FlatFileItemReader">
    <property name="linesToSkip" value="1"/>
    <property name="lineMapper">
        <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
            <property name="lineTokenizer">
                <bean class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
                    <property name="names" value="ID,NAME,DESCRIPTION,PRICE"/>
                </bean>
            </property>
            <property name="fieldSetMapper">
                <bean class="com.way2learnonline.batch.support.ProductFieldSetMapper"/>
            </property>
        </bean>

    </property>
</bean>
```

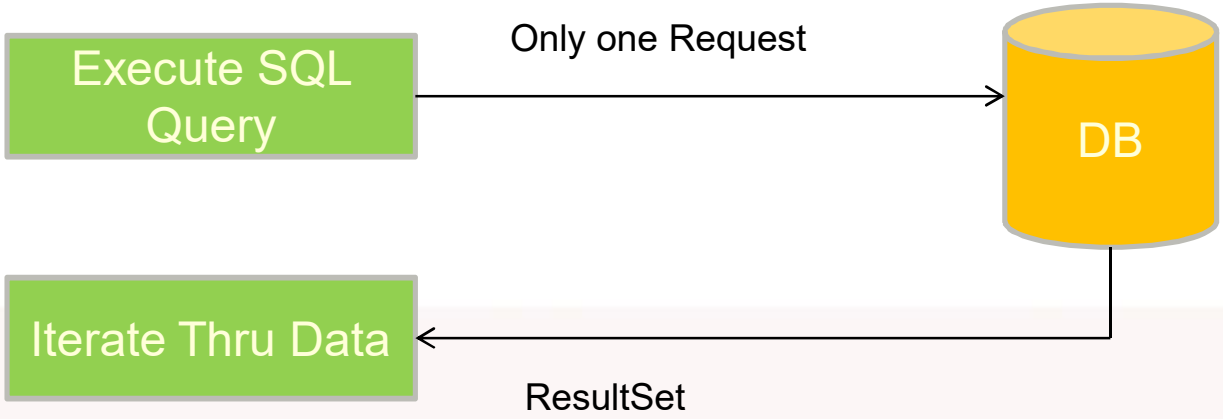
DEMO

Reading from Database

- We will be discussing JDBC based item readers.
- Uses Spring RowMapper interface and JDBC PreparedStatement interface.
- **Two Approaches :**
- Cursor Based
- Paging Based

Cursor Based

- Reading data using ResultSet interface
- It's a simple item reader implementation that opens a JDBC cursor and continually retrieves the next row in the ResultSet.
- Spring Batch relies on JDBC configuration and optimizations to perform efficiently.



Internal Picture

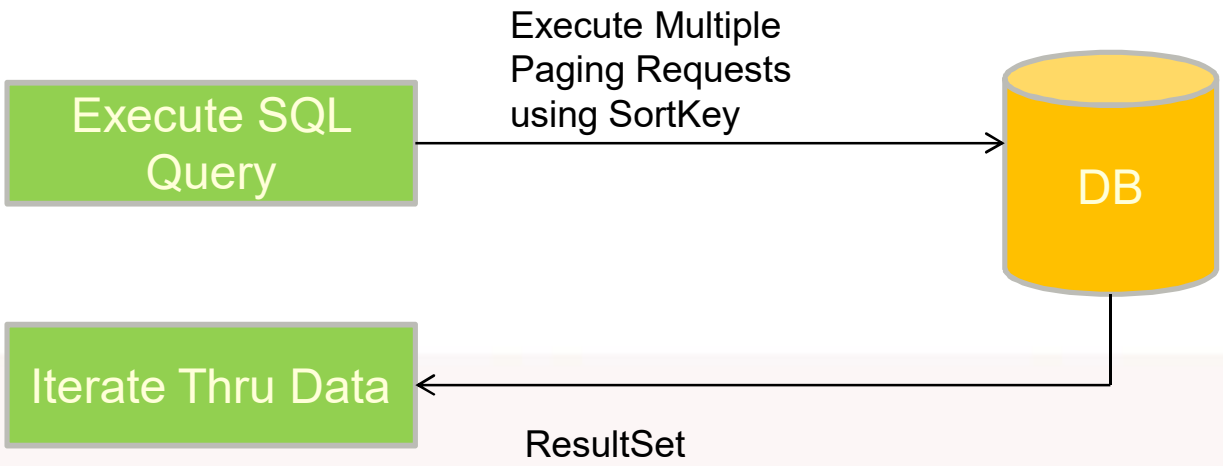
Cursor Based

```
<bean id="productReader" class="org.springframework.batch.item.database.JdbcCursorItemReader">
    <property name="dataSource" ref="dataSource"/>
    <property name="sql" value="select id,name,description,price from productOld"/>
    <property name="rowMapper">
        <bean class="com.way2learnonline.batch.support.ProductRowMapper"/>
    </property>
</bean>
```

DEMO

Paging Based

- In this case, retrieving data consists in successively executing several requests with criteria.
- Spring Batch dynamically builds requests to execute based on a sort key to delimit data for a page.
- To retrieve each page, Spring Batch executes one request to retrieve the corresponding data.



Paging Based

```
<bean id="productFileReader" class="org.springframework.batch.item.database.JdbcPagingItemReader">
    <property name="dataSource" ref="dataSource"/>
    <property name="queryProvider">
        <bean class="org.springframework.batch.item.database.support.SqlPagingQueryProviderFactoryBean">
            <property name="dataSource" ref="dataSource"/>
            <property name="selectClause" value="select id,name,description,price"/>
            <property name="fromClause" value="from productold"/>
            <property name="sortKey" value="id"/>
        </bean>
    </property>
    <property name="pageSize" value="5"/>
    <property name="rowMapper">
        <bean class="com.way2learnonline.batch.support.ProductRowMapper"/>
    </property>
</bean>
```

- Separate properties for selectClause,fromClause whereClause,sortKey
- whereClause is optional
- sortKey is used as start and end of page , and is added to the where clause
- Reason of Separate properties is : Syntax depends on database and Spring batch creates appropriate syntax

Paging Based

- **Choosing a page size**

- There's no definitive value for the page-size setting.

- Lets say the size is 1,000 items
- The page size is usually higher than the commit interval (whose reasonable values range from 10 to 200 items)..

- Remember, the point of paging is to avoid consuming too much memory, so large pages aren't good.

- Small pages aren't good either.
- If you read 1 million items in pages of 10 items (a small page size), you'll send 100,000 queries to the database.

- The good news is that the page size is a parameter in Spring Batch, so you can test multiple values and see which works best for your job.

Which one to use – Cursor vs Paging?

- Cursor-based readers issue one query to the database and stream the data to avoid consuming too much memory.

- Cursor-based readers rely on the cursor implementation of the database and of the JDBC driver.
- Depending on your database engine and on the driver, cursor-based readers can work well . . . or not.

- Page-based readers work well with an appropriate page size
- The trick is to find the best page size for your use case.
-
- With Spring Batch, switching from cursor- to page-based item readers is a matter of configuration and doesn't affect your application code.

- Don't hesitate to test both!

Writers

Writers

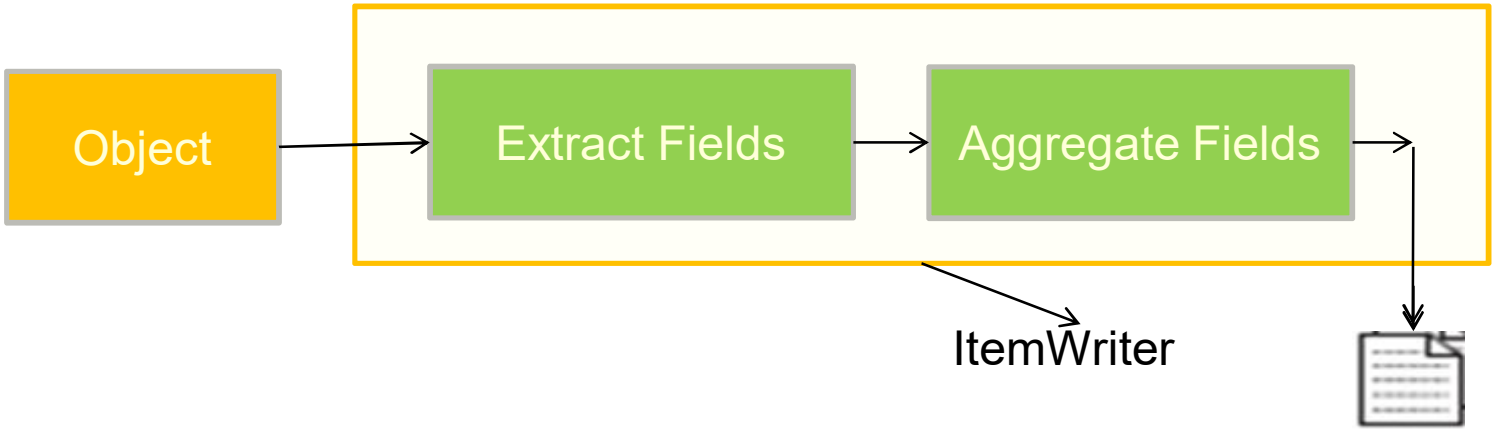
```
public interface ItemWriter<T> {
  void write(List<? extends T> items) throws Exception;
}
```

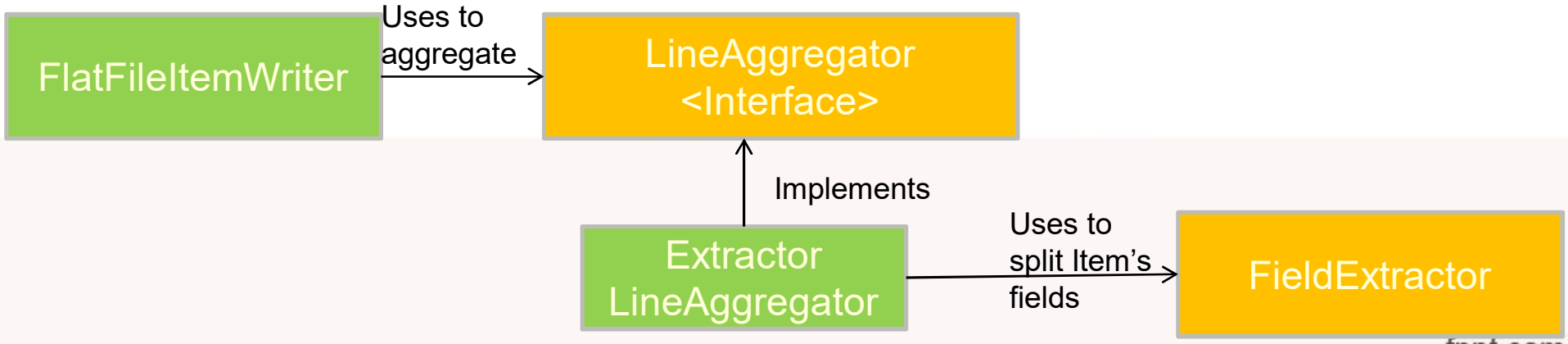
- An ItemReader reads input data
- An ItemProcessor (optionally) processes it,
- And an ItemWriter writes it.
- Spring Batch provides the plumbing to aggregate reading **as per chunk size(commit-interval)** and passing the data to the writer

- Spring Batch has many ItemWriter implementations available :-
- We will discuss few File and Database related Item Writers

Writing Flat Files

- ItemReader used : **FlatFileItemWriter**





Writing Flat Files - 1

```
<!-- Writer declared -->
<!-- The most basic implementation of the LineAggregator interface is the
PassThroughLineAggregator class, which calls toString() on each Product object -->
<!-- It depends on how your toString method is defined in Product class -->

<bean id="productWriter" class="org.springframework.batch.item.file.FlatFileItemWriter">
    <property name="resource" value="file:target/Files/products.txt"/>
    <property name="appendAllowed" value="true"/>
    <property name="lineAggregator">
        <bean class="org.springframework.batch.item.file.transform.PassThroughLineAggregator"/>
    </property>
</bean>
```

DEMO

- Not used any FieldExtractor as of now, will see example next

Writing Flat Files - 2

```
<!-- Writer declared -->
<!--
For more control over the output, you can select which properties to write with the
BeanWrapperFieldExtractor class. This extractor takes an array of property names,
reflectively calls the getters on a source item object, and returns an array of values. -->

<bean id="productWriter" class="org.springframework.batch.item.file.FlatFileItemWriter">
    <property name="resource" value="file:target/Files/products.txt"/>
    <property name="shouldDeleteIfExists" value="true"/>
    <property name="lineAggregator">
        <bean class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
            <property name="fieldExtractor">
                <bean class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor">
                    <property name="names" value="id,name,price"/>
                </bean>
            </property>
        </bean>
    </property>
</bean>
```

Aggregator

Extractor

- A DelimitedLineAggregator converts into a delimited list of strings to be written to file.
- The default delimiter is a comma.

DEMO

Writing Flat Files - 3

```
<!-- Writer declared -->
<!--
WRITING COMPUTED FIELDS
To add computed fields to the output, you create a FieldExtractor implementation. -->

<bean id="productWriter" class="org.springframework.batch.item.file.FlatFileItemWriter">
    <property name="resource" value="file:target/Files/products.txt"/>
    <property name="shouldDeleteIfExists" value="true"/>
    <property name="lineAggregator">
        <bean class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
            <property name="delimiter" value="###"/> <!-- using delimiter other than default delimiter , -->
            <property name="fieldExtractor">
                <bean class="com.way2learnonline.batch.support.ProductFieldExtractor"/>
            </property>
        </bean>
    </property>
</bean>
```

Custom Extractor

DEMO

Writing Flat Files -4

```
<!-- Writer declared -->
<!-- Using FormattingLineAggregator instead of DelimitedLineAggregator. -->
<!-- id output is 9 characters and padded to the left
price output is 6 characters as a float with 2 precision characters
name output is 30 characters and padded to the left

Look at java doc for more info
-->

<bean id="productWriter" class="org.springframework.batch.item.file.FlatFileItemWriter">
    <property name="resource" value="file:target/Files/products.txt"/>
    <property name="shouldDeleteIfExists" value="true"/>
    <property name="lineAggregator">
        <bean class="org.springframework.batch.item.file.transform.FormatterLineAggregator">
            <property name="fieldExtractor">
                <bean class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor">
                    <property name="names" value="id,price,name"/>
                </bean>
            </property>
            <property name="format" value="%-9s%6.2f%-30s"/>
        </bean>
    </property>
</bean>
```

Different Aggregator for Formatted Files

DEMO

Writing to XML

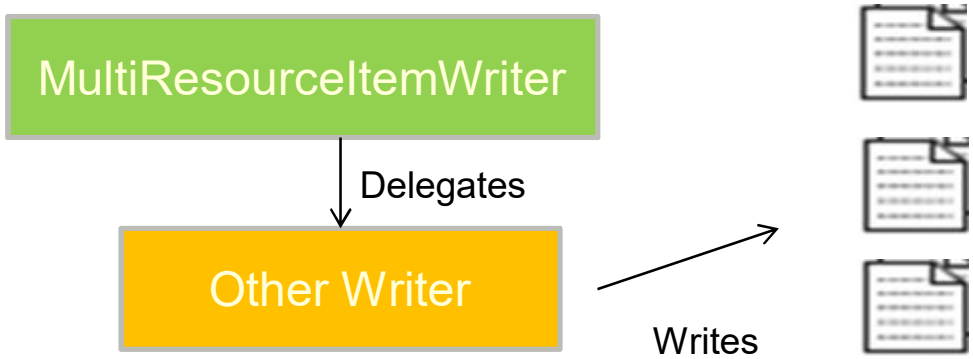
```
<bean id="productWriter" class="org.springframework.batch.item.xml.StaxEventItemWriter">
    <property name="resource" value="file:target/Files/products.xml"/>
    <property name="marshaller" ref="jaxb"/>
    <property name="rootTagName" value="products"/>
</bean>

<bean id="jaxb" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
    <property name="classesToBeBound">
        <list>
            <value>com.way2learnonline.domain.Product</value>
        </list>
    </property>


</bean>
```

- **StaxEventItemWriter** is used.
- This writer uses a marshaller to write XML files.
- A Marshaller is a generic interface provided by the Spring Object/XML Mapping1 module to convert objects to XML, and vice versa.
- Spring OXM supports Java Architecture for XML Binding (JAXB) Castor XML, XMLBeans, JiBX, and XStream.

DEMO

Writing File Sets

- Spring Batch provides a mechanism to write file sets instead of a single file

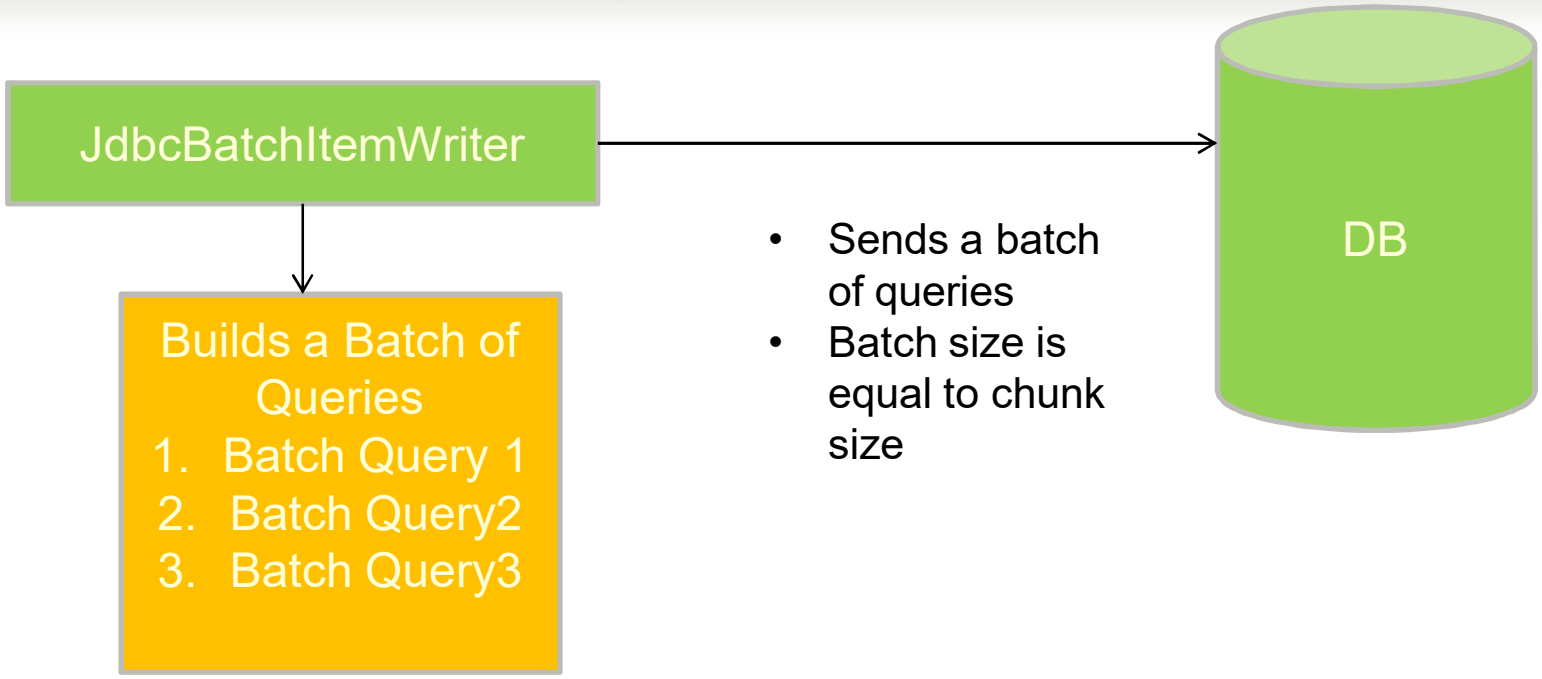



```
<bean id="productWriter" class="org.springframework.batch.item.file.MultiResourceItemWriter">
    <property name="resource" value="file:target/Files/products.txt"/>
    <property name="itemCountLimitPerResource" value="3"/>
    <property name="delegate" ref="productFileWriter"/>
</bean>


<bean id="productFileWriter" class="org.springframework.batch.item.file.FlatFileItemWriter">

    <property name="appendAllowed" value="true"/>
    <property name="lineAggregator">
        <bean class="org.springframework.batch.item.file.transform.PassThroughLineAggregator"/>
    </property>
</bean>
```

Record count per file

DEMO

Writing To Database using *JdbcBatchItemWriter*



- Sends a batch of queries
- Batch size is equal to chunk size

JdbcBatchItemWriter - Way 1

```
<!-- Writer Defined -->
<bean id="productDatabaseWriter" class="org.springframework.batch.item.database.JdbcBatchItemWriter">
    <property name="itemSqlParameterSourceProvider"> <!-- sql parameter values from named parameters -->
        <bean class="org.springframework.batch.item.database.BeanPropertyItemSqlParameterSourceProvider"/><!--  bind item properties to SQL parameter names
    </property>
     <property name="sql" value="INSERT INTO PRODUCT (ID, NAME, PRICE) VALUES(:id, :name, :price)"/>
     <property name="dataSource" ref="dataSource"/>
</bean>
```

- The product bean object is input to the writer
- The values of the bean fields are picked and set to the insert statement
- For this insertSqlParameterSourceProvider and BeanPropertyItemSqlParameterSourceProvider are used

DEMO

JdbcBatchItemWriter - Way 2

```
<!-- Writer Defined -->
<bean id="productDatabaseWriter" class="org.springframework.batch.item.database.JdbcBatchItemWriter">
    <property name="itemPreparedStatementSetter"> <!--using this in place of itemSqlParameterSourceProvider -->
        <bean class="com.way2learnonline.batch.support.ProductWriter"/><!--  bind item properties according to custom class. -->
    </property>
     <property name="sql" value="INSERT INTO PRODUCT (ID, NAME, PRICE) VALUES(?, ?, ?)"/>
     <property name="dataSource" ref="dataSource"/>
</bean>
```

- The product bean object is input to the custom class
- And the custom class sets the value of the **PreparedStatement**

DEMO

CompositeItemWriter

```
<!--
multiple writers for the same chunk. -->

<bean id="productWriter" class="org.springframework.batch.item.support.CompositeItemWriter">
    <property name="delegates">
        <list>
            <ref bean="productFileWriter"/>
            <ref bean="fixedLiengthProductWriter"/>
        </list>

    </property>
</bean>
<bean id="productFileWriter" class="org.springframework.batch.item.file.FlatFileItemWriter">
    <property name="resource" value="file:target/Files/productsDelimited.txt"/>
    <property name="shouldDeleteIfExists" value="true"/>
    <property name="lineAggregator">
        <bean class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
            <property name="fieldExtractor">
                <bean class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor">
                    <property name="names" value="id,name,price"/>
                </bean>
            </property>
        </bean>
    </property>
</bean>
<!--
fixedLiengthProductWriter

-->
<bean id="fixedLiengthProductWriter" class="org.springframework.batch.item.file.FlatFileItemWriter">
    <property name="resource" value="file:target/Files/productsFixed.txt"/>
    <property name="shouldDeleteIfExists" value="true"/>
```

Sending to Two Writers

- Sometimes you need multiple writers for the same chunk.

DEMO

Processing

Processing

```
package org.springframework.batch.item;

public interface ItemProcessor<I, O> {

  O process(I item) throws Exception;

}
```

This doesn't work in some scenarios

Need some processing before ItemWriter

Processing

- **<u>Processing Categories</u>**

- **Transformation**

- 1)The item processor updates read items before sending them to the writer.

- 2)Create a new Object which is of different Type (different from read item)
- Written items may not be of the same type as read items.

- **Filtering**
- The item processor decides whether to send each read item to the writer.

Configuring Processor

Specified Processor

```
<!-- Job defined -->
<!-- processor added here and declared below -->
<batch:job id="readWriteProducts">
    <batch:step id="readWrite">
        <batch:tasklet>
            <batch:chunk reader="productReader" processor="productProcessor" writer="productWriter" commit-interval="3"/>
        </batch:tasklet>
    </batch:step>
</batch:job>


<!-- processor declared -->

<bean id="productProcessor" class="com.way2learnonline.batch.ProductItemProcessor"/>
```

DEMO

Transformation

Case 1)
The item processor updates read items before sending them to the writer.

```
//I am getting an id from datasource but before writing to file , i am generating a new id and setting it to the product
//This id will be used by external system
public class ProductItemProcessor implements ItemProcessor<Product,Product>{


    @Override
    public Product process(Product item) throws Exception {

        String productId=(int)(Math.random()*2000)+"";
        item.setId(productId);
        return item;
    }

}
```

DEMO

Transformation

- 2)Create a new Object which is of different Type (different from read item)
- Written items may not be of the same type as read items.

```
public class ProductToPartnerProductProcessor implements ItemProcessor<Product,PartnerProduct>{

    @Override
    public PartnerProduct process(Product product) throws Exception {

       PartnerProduct partnerProduct = new PartnerProduct();
       partnerProduct.setProdId(product.getId());
       partnerProduct.setProdName(product.getName());
       partnerProduct.setProdDetails(product.getDescription());
       partnerProduct.setProdPrice(product.getPrice().add(new BigDecimal(1000)));
       partnerProduct.setDiscount(20.0);

        return partnerProduct;
    }

}
```

DEMO

Filtering

- Case 3) The item processor decides whether to send each read item to the writer.

```
public class ProductItemProcessor implements ItemProcessor<Product,Product>{

    @Override
    public Product process(Product product) throws Exception {

        if(product.getPrice().intValueExact() < 4000 )
        {
            return null;
        }
        return product;
    }

}
```

- When “null” is returned , item is filtered

DEMO

Composite Item Processors

```
<!-- CompositeItemProcessor delegating to two processors -->

<bean id="productProcessor" class="org.springframework.batch.item.support.CompositeItemProcessor">
    <property name="delegates">
        <list>
            <ref bean="productProcessorForId"/>
            <ref bean="productProcessorForFilter"/>
        </list>
    </property>
</bean>

<bean id="productProcessorForId" class="com.way2learnonline.batch.ProductItemProcessor"/>

<bean id="productProcessorForFilter" class="com.way2learnonline.batch.ProductFilterItemProcessor"/>
```

- When using a composite item processor, the delegates should form a type compatible chain :
- The type of object an item processor returns must be compatible with the type of object the next item processor expects.

DEMO

Java Configuration with Spring Batch

Java Configuration

- Spring Batch has Java configuration support
  - As of Spring Batch 2.2
- Consists of a Java-based builder API
  - Job builder, step builder, etc.
- Benefits
  - Type-safety
  - XML IDE support not longer needed
  - Less verbose than XML

@EnableBatchProcessing

- Registers the Spring Batch infrastructure
- Just needs a DataSource bean

```
@Configuration
@EnableBatchProcessing  //creates job repository , job launcher , transaction manager and batch artifact
@Import(InfrastructureConfig.class)
public class BatchConfig {
```

Declaring the Job

```
@Configuration
@EnableBatchProcessing  //creates job repository , job launcher , transaction manager and batch artifact
@Import(InfrastructureConfig.class)
public class BatchConfig {

    @Autowired
    private JobBuilderFactory jobBuilderFactory;

    @Autowired
    private StepBuilderFactory stepBuilderFactory;


    @Bean
    public Job job(Step firstStep,Step secondStep)
    {
        return jobBuilderFactory.get("UnZipAndreadWriteJob")
                .start(firstStep)
                .next(secondStep)
                //.preventRestart()
                .build();

    }
```

Created by @EnableBatchProcessing

Declares the Job

Configures the job

Creates the Job Bean

Create a Step

```
@Autowired
private StepBuilderFactory stepBuilderFactory;
```

Created by @EnableBatchProcessing

```
@Bean
public Step secondStep(ItemReader<Product> productFileReader,ItemWriter<Product> productJdbcItemWriter)
{
    return stepBuilderFactory.get("readerWriter")
            .<Product,Product>chunk(3)  //need to define generic types of the items handled by the writer, reader, and processor
            .reader(productFileReader)
            .writer(productJdbcItemWriter)
        //  .startLimit(4)
            .build();
}
```

Reader and Writer declared elsewhere and automatically injected

Chunk oriented step created

Create an Item Reader

```
@StepScope
@Bean
public FlatFileItemReader<Product> productFileReader(@Value("#{jobParameters[targetDirectory]}") String targetDirectory,
        @Value("#{jobParameters[targetFile]}") String targetFile) throws MalformedURLException
{


    FlatFileItemReader<Product> reader = new FlatFileItemReader<Product>();
    Resource resource = new UrlResource("file:"+targetDirectory + "/"+ targetFile);
    reader.setResource(resource);

    //FieldSetMapper
    ProductFieldSetMapper fieldSetMapper = new ProductFieldSetMapper();

    //For line to tokens
    DelimitedLineTokenizer lineTokenizer = new DelimitedLineTokenizer();
    lineTokenizer.setDelimiter(",");
    lineTokenizer.setNames(new String[]{"ID", "NAME", "DESCRIPTION", "PRICE"});

    //Creating linemapper and setting fieldset and tokenizer
    DefaultLineMapper<Product> lineMapper = new DefaultLineMapper<Product>();
    lineMapper.setFieldSetMapper(fieldSetMapper);
    lineMapper.setLineTokenizer(lineTokenizer);

    //setting linemapper to reader
    reader.setLineMapper(lineMapper);

    //first line is heading
    reader.setLinesToSkip(1);

    return reader;
}
```

DEMO of Java Config with all XML demos , this is just intro

Execution Context

Requirements

- Case 1)
- There is often a need to pass state between Jobs or Steps, which can be any Java object.
- For example, one Step or Job might be processing records, and a subsequent Job or Step needs to send notification of how many records were processed

- Case 2)
- If there is a failure of record at 6th position and I restart the job. I would like to get it restarted from the same record.

ExecutionContext

- The first option for state transition can be done via the ExecutionContext class.
- This class encapsulates a map of type Map<String, Object>.
- So we are able to store any type of state that needs to be transferred between steps or chunk-oriented item handlers.
- When Step is executed, a **StepExecution** instance is created.
  - Committed at the end of each chunk
- When Job is executed, an instance of **JobExecution** is created.
  - Committed at the end of each step

ExecutionContext

- Via StepExecution.getJobExecution(), we can access the JobExecution instance,
- And via JobExecution.getExecutionContext(), we can access the ExecutionContext instance.
- The StepExecution instance can be injected into any chunk-oriented **ItemProcessor, ItemWriter, and ItemReader** via the initialization method annotated with @BeforeStep
- For **Tasklet**, there is ChunkContext instance as a second parameter of Tasklet.execute().
  - The StepExecution instance can be accessed and then chunkContext.getStepContext().getStepExecution().getExecutionContext()

Case 1 : Sharing State between Two Steps

```
<bean id="promotionListener" class="org.springframework.batch.core.listener.ExecutionContextPromotionListener">
    <property name="keys" value="zipId"/>
</bean>

<!-- Job defined -->

<batch:job id="readWriteProducts">
    <batch:step id="unzip" next="readWrite">
        <batch:tasklet ref="unzipTasklet"> <!-- key is added in StepExecutionContext from tasklet -->
            <batch:listeners>
                <batch:listener ref="promotionListener"/>
            </batch:listeners>
        </batch:tasklet>
    </batch:step>

     <batch:step id="readWrite">
        <batch:tasklet>
            <batch:chunk reader="productFileReader" writer="productDatabaseWriter" commit-interval="3"/>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

Key to be promoted

Promotion Listener

- It requires that a step writes data in its own execution context – StepExecutionContext and that a listener promotes the data to the JobExecutionContext

DEMO

Case 2 : Reading the failed record from same point

```
<!-- Run WITH and WITHOUT incorrectField in products.txt , first time job will fail -
     * few records will get saved in database
    and after removing incorrectField , remaining records will be saved ...but it will honor chunksize -->
    <!-- Any existing spring batch stream that implements ItemStream interface saves the state -->*/

<bean id="productFileReader" class="org.springframework.batch.item.file.FlatFileItemReader">
    <property name="linesToSkip" value="1"/>
    <property name="saveState" value="true"/> <!-- notice saveState as true -->
    <property name="resource" value="classpath:Files/products.txt"/>
    <property name="lineMapper">
        <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
            <property name="lineTokenizer">
                <bean class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
                    <property name="names" value="ID,NAME,DESCRIPTION,PRICE"/>
                </bean>
            </property>
            <property name="fieldSetMapper">
                <bean class="com.way2learnonline.batch.support.ProductFieldSetMapper"/>
            </property>
        </bean>

    </property>
</bean>
```

saveState

- For existing Readers, need to make sure that they implement ItemStream
- Check documentation for the capability of being stateful
- Stateful remembers numbers of lines read by storing the details in jobRepository

DEMO

Case 2 : Custom Item Readers

- Need to implement your own logic for saving record pointer

```
public class DiningReader implements ItemReader<Dining>, StepExecutionListener {
    private ExecutionContext executionContext;
    private int filePosition;
    public Dining read() throws Exception {
        ...
        filePosition++;
        executionContext.put("position", filePosition);
        return dining;
    }

    public void beforeStep(StepExecution stepExecution) {
        this.executionContext = stepExecution.getExecutionContext();
        filePosition = this.executionContext.getInt("position", 0);
    }
...
}
```

Initialize File Position from Last Run

Based on saved position in jobRepository read from file

Also need implementation for afterStep()

Assumes single-threaded step

Introduction to Skip , Restart and Retry

Skip/Retry/Restart

- ***Skip—***
- A line in the flat file is incorrectly formatted.
- Not required to stop the job execution because of a couple of bad lines:
- Configure Spring Batch to skip the line that caused the item reader to throw an exception on a formatting error.
- ***Retry—***
- Some products are already in the database
- Batch updates the database.
- Due to heavy activity the database throws a concurrency exception when the job tries to update a product in a locked row, but retrying the update again a few milliseconds later works.
- You can configure Spring Batch to retry automatically.
- ***Restart—***
- Job fails at a particular record.
- An operator will analyse the input file and correct it before restarting the import.
- Spring Batch can restart the job on the line that caused the failed execution.
- The work performed by the previous execution isn’t lost.

Skip

```
<!-- Job defined -->
<!-- within chunk define skippable exception classes and and skip-limit -->
<batch:job id="readWriteProducts">
    <batch:step id="readWrite">
        <batch:tasklet>
            <batch:chunk reader="productFileReader" writer="productDatabaseWriter" commit-interval="3" skip-limit="4">
                <batch:skippable-exception-classes>
                    <batch:include class="org.springframework.batch.item.file.FlatFileParseException"/>
                </batch:skippable-exception-classes>
            </batch:chunk>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

No. of records to be skipped

Exception to be skipped in include element

3 **DEMO**

Skip

- Listening and Logging skipped items
- Its better to log skipped lines to take actions later on and correct incorrect data
- **Use Listeners**

```
public interface SkipListener<T,S> extends StepListener {
  void onSkipInRead(Throwable t);
  void onSkipInProcess(T item, Throwable t);
  void onSkipInWrite(S item, Throwable t);
}
```

- Either create a class that implement this interface
- Or use @OnSkipInRead, @OnSkipInProcess, and @OnSkipInWrite

Skip

```
<!-- Job defined -->
<!-- Using Skip Listener to log error-->
<batch:job id="readWriteProducts">
    <batch:step id="readWrite">
        <batch:tasklet>
            <batch:chunk reader="productFileReader" writer="productDatabaseWriter" commit-interval="3" skip-limit="4">
                <batch:skippable-exception-classes>
                    <batch:include class="org.springframework.batch.item.file.FlatFileParseException"/>
                </batch:skippable-exception-classes>
            </batch:chunk>
            <batch:listeners>
                <batch:listener ref="skipListener"/>
            </batch:listeners>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

Batch-config4.xml

Specified Listener

```
public class DatabaseSkipListener {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @OnSkipInRead
    public void log(Throwable throwable) {
        if(throwable instanceof FlatFileParseException) {
            FlatFileParseException exception = (FlatFileParseException) throwable;
            jdbcTemplate.update(
                "insert into skipped_product " +
                "(line,line_number) values (?,?)",
                exception.getInput(),exception.getLineNumber()
            );
    }
    }

}
```

DEMO

Restart

- Jobs defined via batch namespace considered restartable after failure by default
  - Disable with restartable="false"
- Requires execution context state to be persisted
  - Otherwise Executions would always start from the beginning
- Spring Batch checks for existing job execution
  - If found , it's a restart
  - If not , it's a new start
- When Batch executes the Step with or without the same JobParameters and it completes with RepeatStatus.FINISHED, its not started again.
- This behaviour can be changed by setting the XML attribute allow-start-if-complete of <tasklet> to true, when we build Step.
- We can also define the number of times that Step can be executed. This can be configured via the start-limit XML attribute of <tasklet> during Step construction.

DEMO

Retry

- A processing error could be transient
- Simply retrying could make the operation successful.
- Can mark certain exceptions retryable.

```
  <job id="simpleRecordsJob">
    <step id="simpleRecordsStep">
      <tasklet>
        <chunk reader="simpleRecordReader" writer="simpleRecordWriter"
          processor="simpleRecordProcessor" commit-interval="4" retry-limit="3">
          <retryable-exception-classes>
            <include class="java.lang.IllegalStateException"/>
          </retryable-exception-classes>
          <retry-listeners>
            <listener ref="simpleRetryListener"/>
          </retry-listeners>
        </chunk>
      </tasklet>
    </step>
  </job>
</beans:beans>
```

- Directly implement the **RetryListener** interface;
- Defines two lifecycle methods—open(first attempt of retry) and close(final attempt of retry) and **onError(** Called after every unsuccessful attempt at a retry.)
- Methods open and close : not used much.
- A better way is to extend the **RetryListenerSupport** class and override the **onError** method

Scaling and Parallel Processing

Scaling

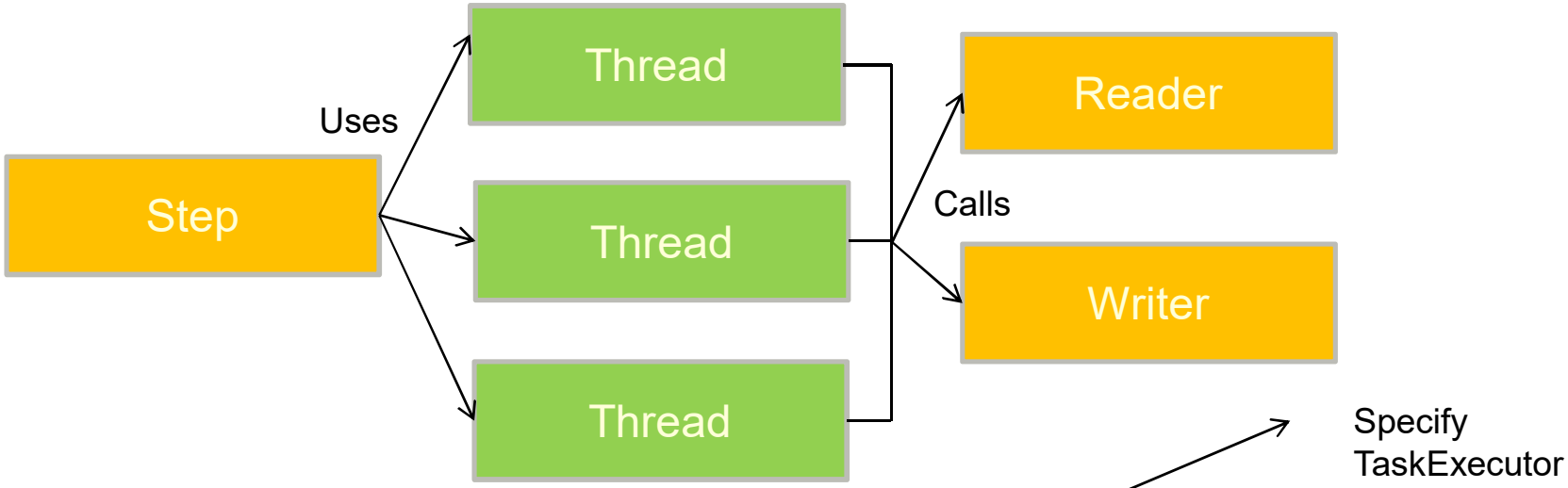
- **Scaling Strategies :-**

- **Multithreaded Step**
- A step is multithreaded

- **Parallel Step**
- Steps in parallel using multithreading

- **Partitioned Step**
- Partitions data and splits up processing-379

MultiThreaded Step

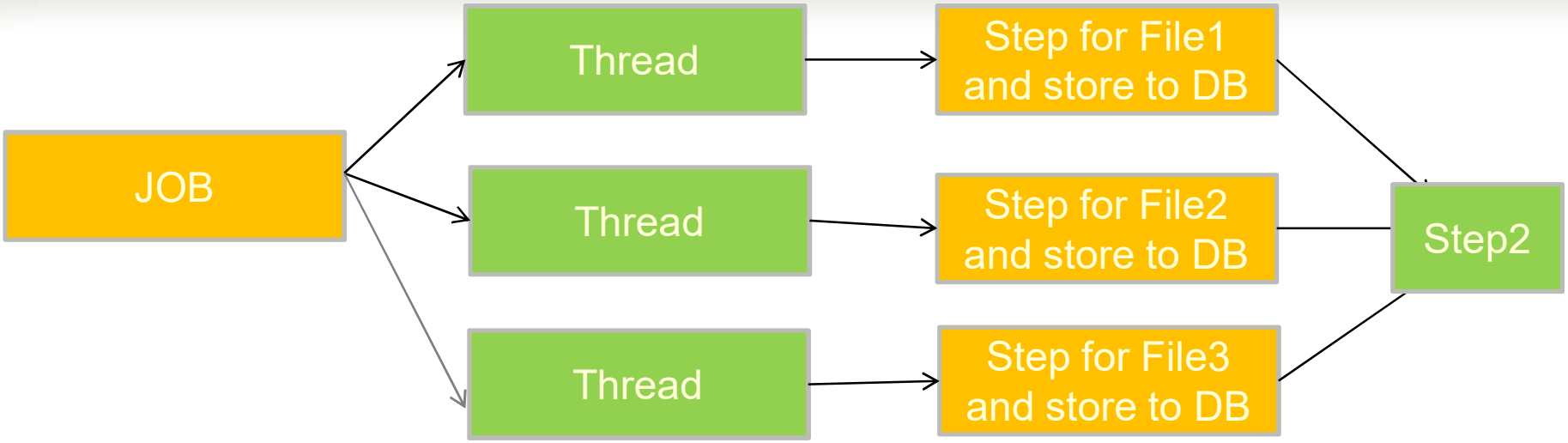



```
  <job id="simpleRecordsJob">
    <step id="simpleRecordsStep">
      <tasklet task-executor="taskExecutor">
        <chunk reader="simpleRecordReader" writer="simpleRecordWriter"
          processor="simpleRecordProcessor" commit-interval="4" />
      </tasklet>
    </step>
  </job>

  <task:executor id="taskExecutor" pool-size="10"/>
</beans:beans>
```

MultiThreaded Step

- No guarantee of the item processing order.
- Check the documentation of the readers and writers you use before configuring a step for multithreading.
- Most of the built-in Spring Batch readers and writers aren’t thread-safe and therefore are unsafe for use in a multithreaded step.
- What would you do if there is such a need?
- Write a custom thread safe Reader/Writer class that synchronizing delegator for the Built-in reader/writer.
- There could be other workarounds available , refer documentation.

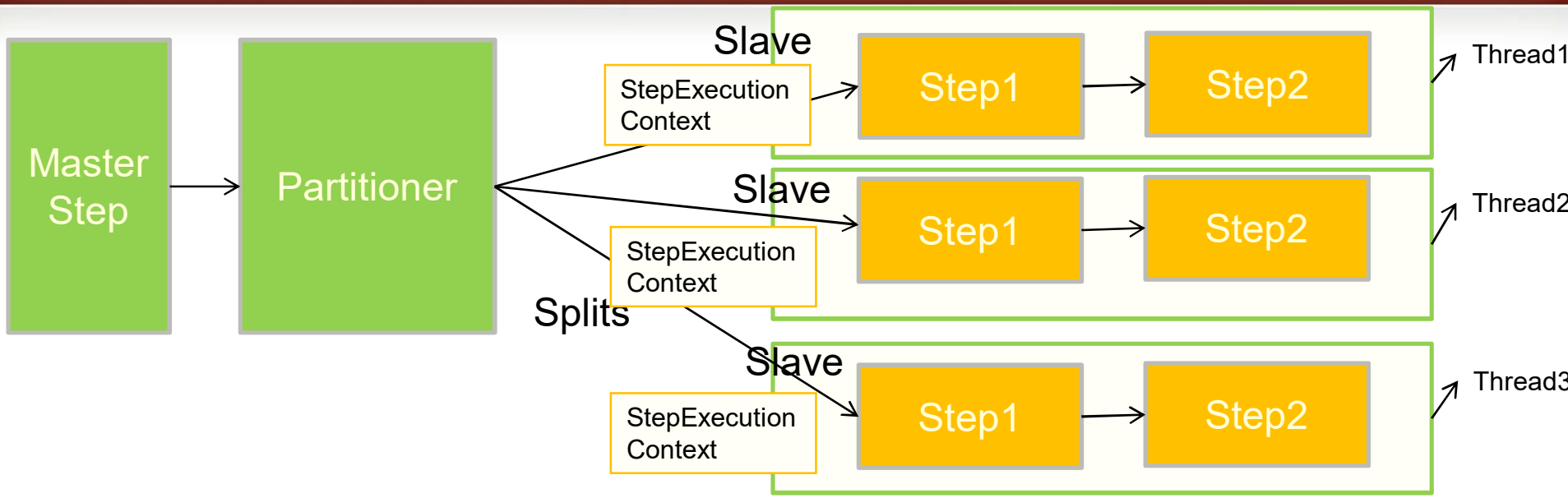
Parallel Step




```
<batch:job id="importProductsJob">
  <batch:step id="decompress" next="readWrite">
    <batch:tasklet ref="decompressTasklet"/>
  </batch:step>
  <batch:split id="readWrite" next="moveProcessedFiles">
    <batch:flow>
      <batch:step id="readWriteBookProduct"/>
    </batch:flow>
    <batch:flow>
      <batch:step id="readWriteMobileProduct"/>
    </batch:flow>
  </batch:split>
  <batch:step id="moveProcessedFiles">
    <batch:tasklet ref="moveProcessedFilesTasklet" />
  </batch:step>
</batch:job>
```

- Speeds up overall job execution
- When there are **independent** steps

Partitioned Step



- With a partitioned step you actually have complete distinct StepExecutions.
- Each StepExecution works on it's own partition of the data.
- This way the reader does not have problems reading the same data because each reader is only looking at a specific slice of data.

Partitioned Step

```
<batch:job id="readWriteProducts" xmlns="http://www.springframework.org/schema/batch">
    <batch:step id="masterStep">
        <!-- master step, 4 threads (grid-size)  -->
        <!-- partitioned into multiple slave steps where each has its own step execution context -->
        <batch:partition step="slaveStep" partitioner="partitioner">
            <batch:handler grid-size="4" task-executor="taskExecutor"/>
        </batch:partition>
    </batch:step>
</batch:job>

<batch:step id="slaveStep" xmlns="http://www.springframework.org/schema/batch">
        <batch:tasklet>
            <batch:chunk reader="productReader" writer="productWriter" commit-interval="3"/>
        </batch:tasklet>
</batch:step>
```

DEMO