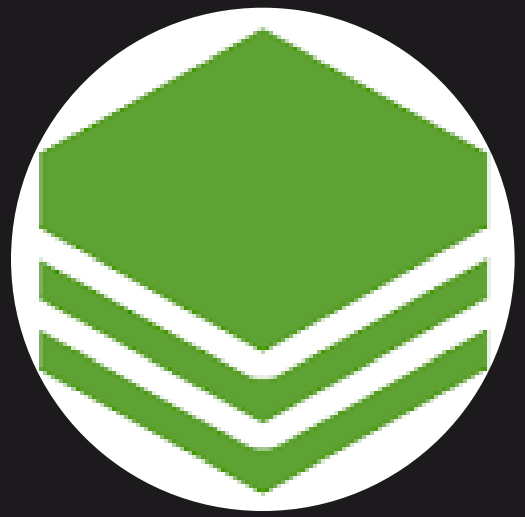


Spring Batch

A lightweight, comprehensive batch framework designed to enable the development of robust batch applications vital for the daily operations of enterprise systems.

Efficient Use of Resources

Why Batch Processing?

- Many applications within the enterprise domain require bulk processing to perform business operations in mission critical environments. These business operations include:
  - Automated, complex processing of large volumes of information that is most efficiently processed without user interaction. These operations typically include time-based events (such as month-end calculations, notices, or correspondence).
  - Periodic application of complex business rules processed repetitively across very large data sets (for example, insurance benefit determination or rate adjustments).
  - Integration of information that is received from internal and external systems that typically requires formatting, validation, and processing in a transactional manner into the system of record. Batch processing is used to process billions of transactions every day for enterprises.

Spring Batch

- Spring Batch provides reusable functions that are essential in processing large volumes of records, including logging/tracing, transaction management, job processing statistics, job restart, skip, and resource management.

- It also provides more advanced technical services and features that will enable extremely high-volume and high performance batch jobs through optimization and partitioning techniques.

- Simple as well as complex, high-volume batch jobs can leverage the framework in a highly scalable manner to process significant volumes of information.

Spring Batch

- Spring Batch is not a scheduling framework.
- There are many good enterprise schedulers (such as Quartz, Tivoli, Control-M, etc.) available in both the commercial and open source spaces.
- It is intended to work in conjunction with a scheduler, not replace a scheduler.

Input Data
Large amount of structured data
Commonly file-based

Job
Runs periodically to process input data efficiently

Output
Processed data is written, often to a database or other job

What is Batch Processing?

Batch Processing, … is defined as the processing of data without interaction or interruption

JSR-352 introduces an exciting new java specification for building , deploying and running batch application

Scenarios for Batch Processing?

- A typical batch program generally:
  - Reads a large number of records from a database, file, or queue.
  - Processes the data in some fashion.
  - Writes back data in a modified form.

- Spring Batch automates this basic batch iteration, providing the capability to process similar transactions as a set, typically in an offline environment without any user interaction.
- Batch jobs are part of most IT projects, and Spring Batch is the only open source framework that provides a robust, enterprise-scale solution.

Business Scenarios

- Commit batch process periodically
- Concurrent batch processing: parallel processing of a job
- Staged, enterprise message-driven processing
- Massively parallel batch processing
- Manual or scheduled restart after failure
- Sequential processing of dependent steps (with extensions to workflow-driven batches)
- Partial processing: skip records (for example, on rollback)
- Whole-batch transaction, for cases with a small batch size or existing stored procedures/scripts

Technical Objectives

- Batch developers use the Spring programming model: Concentrate on business logic and let the framework take care of infrastructure.
- Clear separation of concerns between the infrastructure, the batch execution environment, and the batch application.
- Provide common, core execution services as interfaces that all projects can implement.
- Provide simple and default implementations of the core execution interfaces that can be used 'out of the box'.
- Easy to configure, customize, and extend services, by leveraging the spring framework in all layers.
- All existing core services should be easy to replace or extend, without any impact to the infrastructure layer.
- Provide a simple deployment model, with the architecture JARs completely separate from the application, built using Maven.

Batch Data Processing?

- **Benefits**
  - Successfully handle high volumes of data
  - can stop and restart job at any time
  - Mature approach

- **Challenges**
  - Finite input data set
  - Can impact real-time processing
  - overkill for simpler computations)

Spring Batch Vitals

- Released in 2008
- Takes Advantage of Spring framework
- Current Version is 4.1
- Fully Annotation Driven from 4.0

Spring Batch Benefits

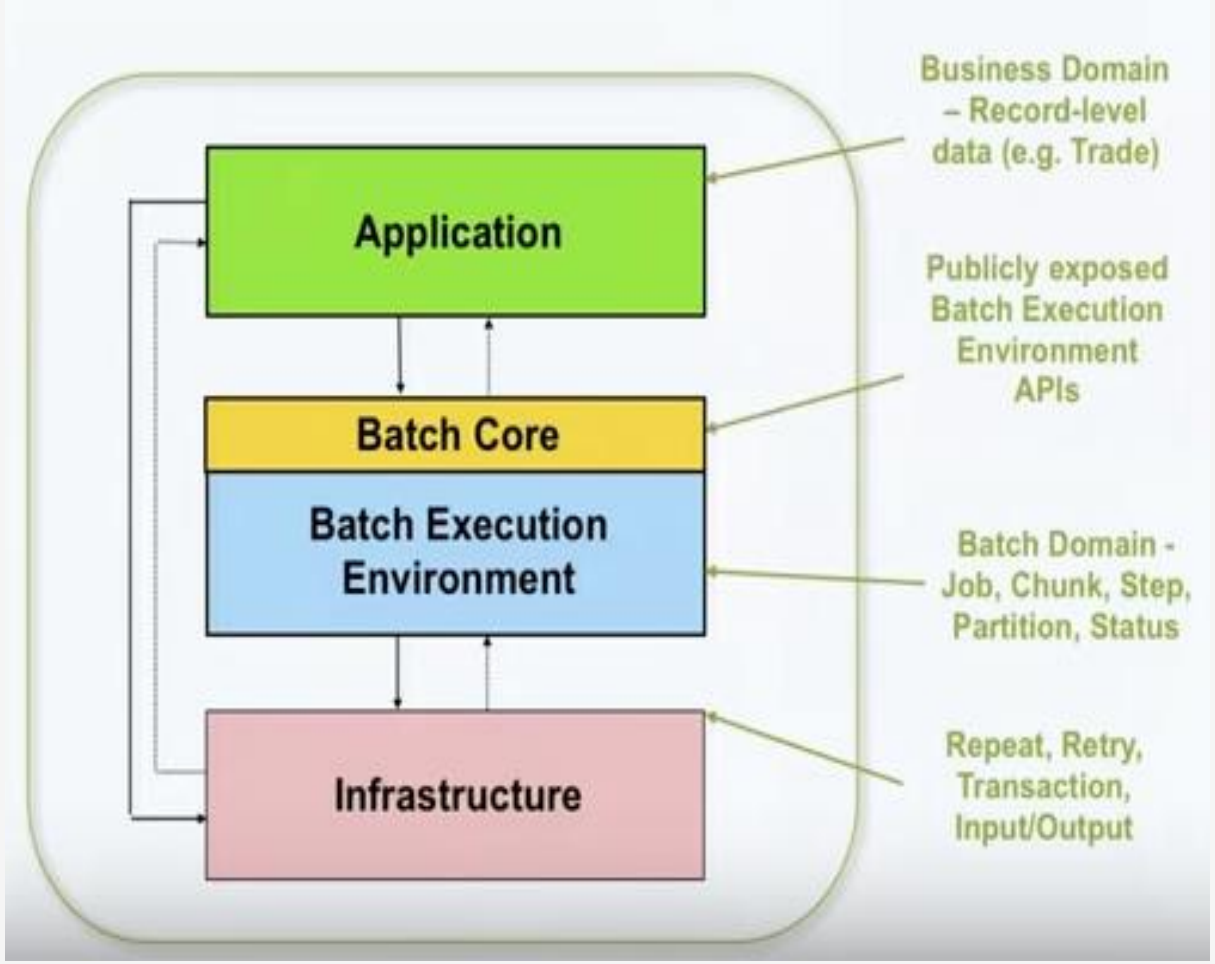
- Spring Based
- Java Based
- Open Source
- Support for Batch Features

Summary so far…

- History of Batch Processing
- Basic Definition of batch processing
- Batch usage Scenario
- Basic definition of Stream Processing
- Benefits and challenges of Batch Processing
- Spring Batch

Spring Batch Architecture & Domain Language of Spring Batch

Layered Architecture



Batch Configuration

- **Spring Batch supports configuration using**
  - **batch Namespace**
  - **Java Based Configuration**

Dependencies

Batch Namespace

- The batch namespace include the following tags.
  - <batch:job-repository>
  - <batch:job>
  - <batch:job-listener>
  - <batch:step>
  - <batch:step-listener>
  - <batch:flow>

Job-Repository Configuration

- Configure JobRepository
  - The job repository is a key feature of the Spring Batch infrastructure because it provides information about batch processing.
  - it saves information about the details of job executions.
  - Spring Batch provides two kinds of DAOs at this level:
    - In-memory with no persistence
    - Persistent with metadata using JDBC

In-Memory Job Repository

```
<bean id="jobRepository"
        class="org.springframework.batch.core.repository
        .support.MapJobRepositoryFactoryBean">
        <property name="transactionManager"
                ref="transactionManager"/>
</bean>

<bean id="transactionManager"
        class="org.springframework.batch.support
        .transaction.ResourcelessTransactionManager"/>
```

Persistent Job Repository

<batch:job-repository/>

- Attributes
  - data-source
  - transaction-manager
  - isolation-level-for-create
  - max-varchar-length
  - table-prefix
  - lob-handler

Jobs &Steps Configuration

- Spring Batch XML configures batch components such as job, step, tasklet, and chunk, as well as their relationships. Together, all these elements make up batch processes.

- When implementing a batch application with Spring Batch, the top-level entity is the job, and it's the first entity you configure when defining a batch process.

- To configure a job, you use the Spring Batch XML **job** element.

Jobs &Steps Configuration

```
<batch:job id="hellojob">
        […]
</batch:job>
```

- Job attributes
  - id
  - restartable
  - incrementer
  - abstract
  - parent
  - job-repository

Jobs &Steps Configuration

- Steps define the sequence of actions a job will take, one at a time.
- You configure a job step using the step element.
- Configuring a step is simple because a step is a container for a tasklet, executed at a specific point in the flow of a batch process.
- Attributes
  - id
  - next
  - parent
  - abstract

Jobs &Steps Configuration

```
<job id="importProductsJob">
        <step id="decompress" next="readWrite">
                [...]
        </step>
        <step id="readWrite">
                [...]
        </step>
</job>
```

Configuring tasklets and chunks

- The tasklet and chunk are step elements used to specify processing.

TASKLET

- A tasklet corresponds to a transactional, potentially repeatable process occurring in a step. You can write your own tasklet class by implementing the Tasklet interface or use the tasklet implementations provided by Spring Batch.

- To configure a tasklet, define a tasklet element within a step element.

Configuring tasklets and chunks

CHUNK-ORIENTED TASKLET

- Spring Batch provides a tasklet class to process data in chunks: the ChunkOriented Tasklet.

- You typically use chunk-oriented tasklets for read-write processing. In chunk processing, Spring Batch reads data chunks from a source and transforms, validates, and then writes data chunks to a destination.

- The chunk child element of the tasklet element configures chunk processing.

- Chunk attributes
  - reader
  - processor
  - writer
  - commit-interval

Configuring tasklets and chunks

```
<batch:job id="importProductsJob">
        [...]
        <batch:step id="readWrite">
                <batch:tasklet>
                        <batch:chunk
                reader="productItemReader"
                processor="productItemProcessor"
                writer="productItemWriter"
                commit-interval="100"/>
                </batch:tasklet>
        </batch:step>
</batch:job>
```

Running batch jobs

- **Running jobs from the command line**
- **Scheduling jobs**
- **Embedding Spring Batch in a web application**

- Launching a Spring Batch job is easy because the framework provides a Java-based API for this purpose.

- However, how you call this API is another matter and depends on your system. Perhaps you'll use something simple like the cron scheduler to launch a Java program.

- Alternatively, you may want to trigger your jobs manually from a web application

Spring Batch launcher API

- The heart of the Spring Batch launcher API is the JobLauncher interface.

```
ApplicationContext context = (…)
JobLauncher jobLauncher =
context.getBean(JobLauncher.class);
Job job = context.getBean(Job.class);
jobLauncher.run(
        job,
        new JobParametersBuilder()
        .addString("inputFile", "file:./products.txt")
        .addDate("date", new Date())
        .toJobParameters()
);
```

Synchronous vs. asynchronous launches

- By default, the JobLauncher run method is synchronous: the caller waits until the job execution ends (successfully or not).
- Synchronous launching is good in some cases: if you write a Java main program that a system scheduler like cron launches periodically, you want to exit the program only when the execution
- But imagine that an HTTP request triggers the launching of a job.

Synchronous vs. asynchronous launches

- To make the job launcher asynchronous, just provide it with an appropriate Task-Executor, as shown in the following snippet:

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

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

Launching from the command line

- Spring Batch provides the CommandLineJobRunner class to launch jobs.
- This launcher should remove any need for custom command-line launchers because of its flexibility.

- mvn dependency:copy-dependencies –DoutputDirectory=lib

- The first parameter to the CommandLineJobRunner is the location of the Spring configuration file, and the second parameter is the name of the Job (the name of the corresponding Spring bean).

- you specify job parameters after the name of the job, using the name=value syntax

Launching from the command line

- **LAUNCHING WITHOUT JOB PARAMETERS**
  - The simplest use of the CommandLineJobRunner is to launch a job that doesn't require any parameters.

```
java -classpath "./lib/*"
org.springframework.batch.core.launch.support.CommandLineJobRunner
import-products-job.xml importProductsJob
```

- **LAUNCHING WITH JOB PARAMETERS**

```
java -classpath "./lib/*"
org.springframework.batch.core.launch.support.CommandLineJobRunner
        import-products-job.xml importProductsJob
        inputFile=file:./products.txt date=2019/12/08
```

Job schedulers

- A job scheduler is a program in charge of periodically launching other programs, in our case, batch processes.

- Spring scheduler
  - The Spring framework scheduler; configurable with XML or annotations, it supports cron expressions; available in Spring 3.0 and later
- Quartz (http://www.quartz-scheduler.org/)
  - What is the Quartz Job Scheduling Library? Quartz is a richly featured, open source job scheduling library that can be integrated within virtually any Java application - from the smallest stand-alone application to the largest e-commerce system.
- Cron
  - A job scheduler available on UNIX-like systems; uses cron expressions to periodically launch commands or shell scripts

Spring scheduler

- As of version 3.0, the Spring Framework offers a declarative way to schedule jobs without requiring extra dependencies for your Spring Batch jobs, because Spring Batch sits on top of Spring.

- Spring's lightweight scheduling provides features like cron expressions, customization of threading policy, and declarative configuration with XML or annotations.

- For XML configuration, Spring provides the task namespace XML vocabulary

Spring scheduler

- **Steps to use the Spring scheduler:**

- Set up the scheduler.
  - This is where you decide whether or not to use a thread pool.
  - This setup is optional, and Spring uses a single-threaded scheduler by default.

- Set up the Java methods to launch periodically.
  - You can use XML or annotations on the target methods. In our case, those methods use the Spring Batch API to launch jobs.

Spring scheduler

SCHEDULING OPTIONS

- Your scheduling requirements can be as simple as "every minute" or as complex as "the last weekday of the month at 23:00."
- Spring allows you to do that by supporting cron expressions—with its own engine—but also lets you trigger a job at a fixed rate without resorting to cron expressions

Scheduling option	XML attribute	Annotation attribute	Description
Fixed rate	fixed-rate	fixedRate	Launches periodically, using the *start* time of the previous task to measure the interval
Fixed delay	fixed-delay	fixedDelay	Launches periodically, using the *completion* time of the previous task to measure the interval
Cron	cron	cron	Launches using a cron expression

Spring scheduler

SCHEDULER SETUP

- Spring uses a dedicated bean to schedule jobs. You can declare this bean using the task namespace prefix:

- **<task:scheduler id="scheduler" />**

- NOTE Remember that declaring a scheduler is optional. Spring uses the default single-threaded scheduler as soon as you declare scheduled tasks.

- Even though Spring uses reasonable defaults, declaring a scheduler explicitly is good practice because it reminds you that an infrastructure bean takes care of the actual scheduling.

- It also serves as a reminder that you can tweak this scheduler to use a thread pool:

- **<task:scheduler id="scheduler" pool-size="10" />**

- Multiple threads are useful when you need to schedule multiple jobs and their launch times overlap. You don't want some jobs to wait because the single thread of your scheduler is busy launching another job.

Spring scheduler

- **SCHEDULING WITH XML**

```
public class SpringSchedulingLauncher {
        private Job job;
        private JobLauncher jobLauncher;

        public void launch() throws Exception {
        JobParameters jobParams = createJobParameters();
        jobLauncher.run(job, jobParams);
        }
        private JobParameters createJobParameters() {
        }
        // setters
}
```

Spring scheduler

- **Scheduling with Spring and XML**

```
<bean id="springSchedulingLauncher"
class="com.pratap.SpringSchedulingLauncher">
        <property name="job" ref="job" />
        <property name="jobLauncher" ref="jobLauncher" />
</bean>


<task:scheduler id="scheduler" />

<task:scheduled-tasks scheduler="scheduler">
        <task:scheduled ref="springSchedulingLauncher"
                method="launch"
                fixed-rate="1000" />
</task:scheduled-tasks>
```

Spring scheduler

- **SCHEDULING WITH ANNOTATIONS**

```
public class SpringSchedulingAnnotatedLauncher {
        private Job job;
        private JobLauncher jobLauncher;

        @Scheduled(fixedRate=1000)
        public void launch() throws Exception {
        JobParameters jobParams = createJobParameters();
        jobLauncher.run(job, jobParams);
        }

        private JobParameters createJobParameters() {

        }
        //setters
}
```

Spring scheduler

- **SCHEDULING WITH ANNOTATIONS**

When using the @Scheduled annotation, the Java class does part of the configuration itself.

The XML configuration is shorter, but you need to tell Spring to look for @Scheduled annotations with the task:annotation-driven element,

```
<bean id="springSchedulingAnnotatedLauncher"
        class="com.pratap.SpringSchedulingAnnotatedLauncher">
        <property name="job" ref="job" />
        <property name="jobLauncher" ref="jobLauncher" />
</bean>

<task:scheduler id="scheduler" />

<task:annotation-driven scheduler="scheduler" />
```

Quartz

- Job scheduling library

- Components of quartz
  - Job
  - Trigger
    - Simple trigger
    - cron trigger
  - Scheduler

  - <!--https://mvnrepository.com/artifact/org.quartz-scheduler/quartz -->
  - <dependency>
  - <groupId>org.quartz-scheduler</groupId>
  - <artifactId>quartz</artifactId>
  - <version>2.3.0</version>
  - </dependency>

Quartz

```
public class QuartzJob implements Job{
        public void execute(JobExecutionContext args){

                System.out.println("Hello");
                System.out.println(new Date());
        }
}
```

Quartz

```
public class QuartzMain{
        public void void main(String []args){

         JobDetail job = JobBuilder.newJob(QuartzJob.class);

         Trigger t1 = TriggerBuilder.newTrigger()
                              .withIdentity("simpletrigger")
                              .startNow()
                              .build();

         Scheduler sc = StdSchedulerFactory.getDefaultScheduler();

         sc.start();
         sc.scheduleJob(job , t1 );
        }
}
```

Quartz

```
public class QuartzMain{
        public void void main(String []args){

         JobDetail job = JobBuilder.newJob(QuartzJob.class);

         Trigger t1 = TriggerBuilder.newTrigger()
                                    .withIdentity("CronTrigger")

        .withSchedule(CronScheduleBuilder.cronSchedule("0 0/1 * 1/1 * ? *"));
                                    .build();

         Scheduler sc = StdSchedulerFactory.getDefaultScheduler();

         sc.start();
         sc.scheduleJob(job , t1 );
        }
}
```

http://www.cronmaker.com/

www.Crontab.guru

Quartz

- Quartz has three main components: **a scheduler, a job, and a trigger.**

- A scheduler, which is obtained from a SchedulerFactory, serves as a registry of JobDetails (a reference to a Quartz job) and triggers and it is responsible for executing a job when its associated trigger fires.

- A job is a unit of work that can be executed. A trigger defines when a job is to be run. When a trigger fires, telling Quartz to execute a job, a JobDetails object is created to define the individual execution of the job.

- In order to integrate Quartz with your Spring Batch process, you need to do the following:

  - Add the required dependencies to your pom.xml file.
  - Write your own Quartz job to launch your job using Spring's QuartzJobBean.
  - Configure a JobDetailBean provided by Spring to create a Quartz JobDetail.
  - Configure a trigger to define when your job should run

Quartz

- Dependencies
  - The first is the Quartz framework itself. The second dependency we add, is for the spring-context-support artifact. This package from Spring, provides the classes required to integrate Quartz easily with Spring.

Reading Data

- Spring Batch can use different data sources as input to batch processes.

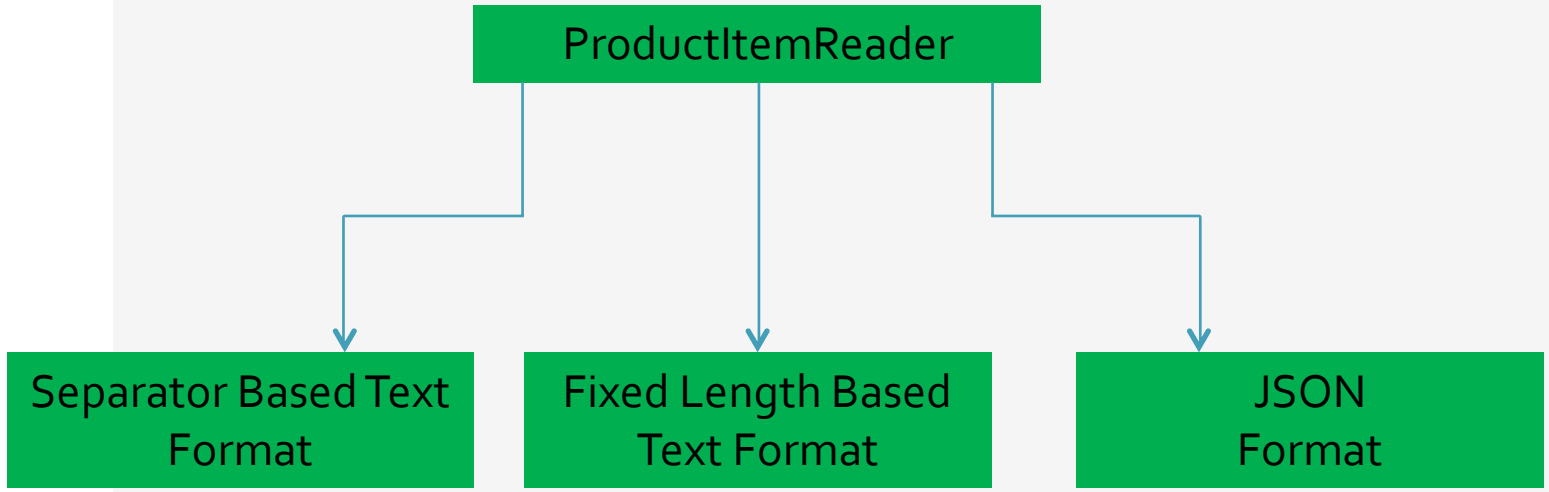
- Data sources correspond to flat files, XML, and JavaScript Serialized Object Notation (JSON).

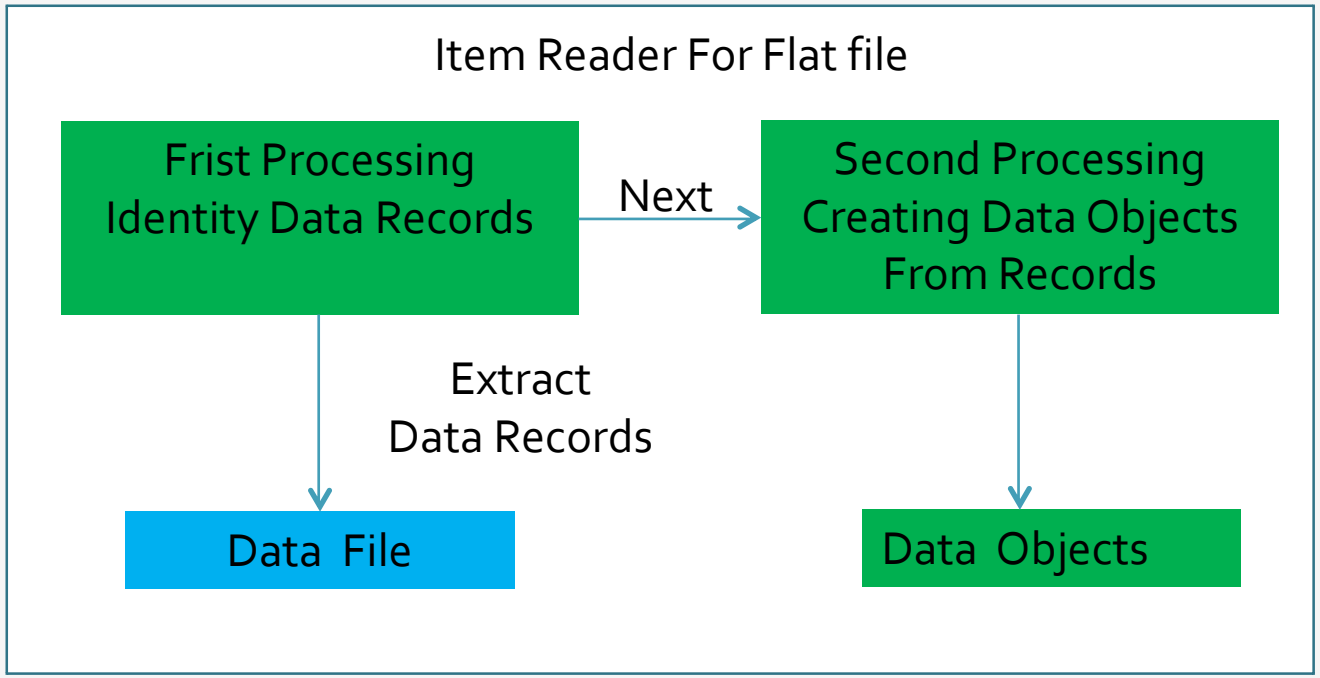
- Spring Batch also supports other types of data sources, such as Java Message Service (JMS), in the message-oriented middleware world.

- **Data reading concepts**

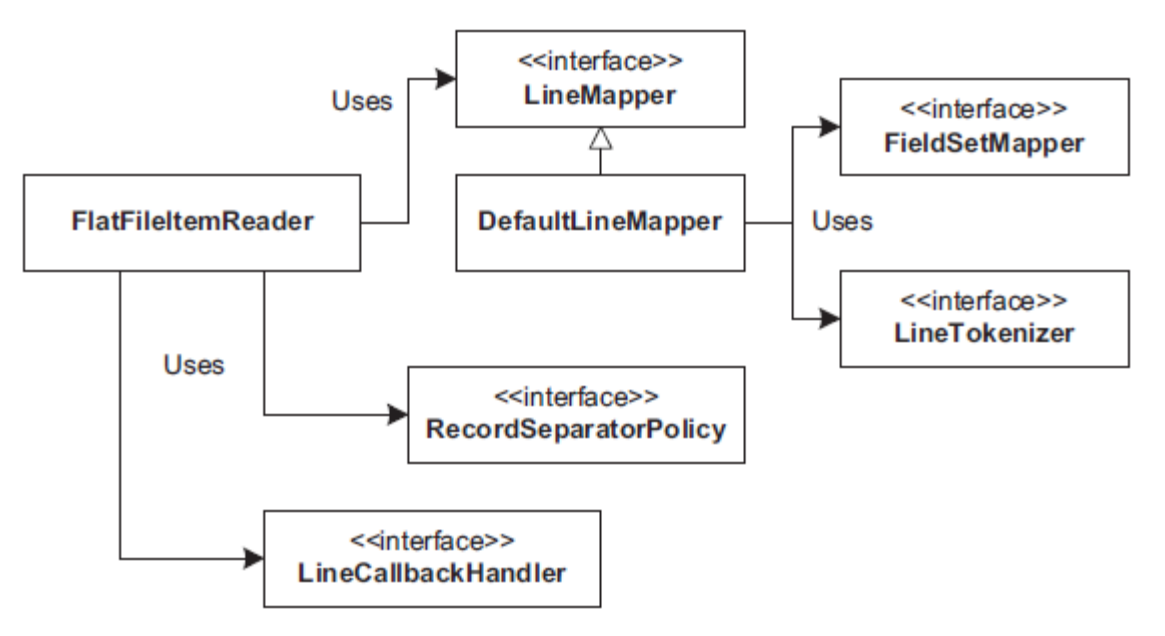
```
public interface ItemReader<T> {
T read() throws Exception, UnexpectedInputException,
            ParseException, NonTransientResourceException;
}
```

Reading Flat File





Reading Flat File



Configuring FlatFileItemReader

```
<bean id="productItemReader"
class="org.springframework.batch.item.file.FlatFileItemReader">
<property name="resource" value="datafile.txt"/>
<property name="linesToSkip" value="1"/>
<property name="recordSeparatorPolicy"
ref="productRecordSeparatorPolicy"/>
<property name="lineMapper" ref="productLineMapper"/>
</bean>
<bean id="productRecordSeparatorPolicy" class="(...)">
[...]
</bean>
```

Extracting character-separated fields

```
<bean id=" productLineTokenizer"
        class="org.springframework.batch.item.file
        .transform.DelimitedLineTokenizer">
<property name="delimiter" value=","/>
<property name="names"
        value="id,name,description,price"/>
</bean>
```

EXTRACTING FIXED-LENGTH FIELDS

```
<bean id=" productLineTokenizer"
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>
```

Reading JSON

Spring Batch provides support for JSON with a LineMapper implementation called JsonLineMapper

Configuring the JsonLineMapper class is simple because line parsing is built into the class, and each FieldSet maps to a java.util.Map.

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

JsonLineMapper Wrapper

```
public class JsonLineMapperWrapper implements LineMapper<Product> {
private JsonLineMapper delegate;
public Product mapLine(String line, int lineNumber) {
Map<String,Object> productAsMap
= delegate.mapLine(line, lineNumber);
Product product = new Product();
product.setId((String)productAsMap.get("id"));
product.setName(
(String)productAsMap.get("name"));
product.setDescription(
(String)productAsMap.get("description"));
product.setPrice(
new Float((Double)productAsMap.get("price")));
return product;
}
}
```

Reading XML files

StaxEventItemReader

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
        <user>
                <id>1</id>
                <name>pratap</name>
                <dept>001</dept>
                <salary>10000</salary>
        </user>
        <user>
                <id>2</id>
                <name>prasanth</name>
                <dept>002</dept>
                <salary>15000</salary>
        </user>
</users>
```

Reading XML files

```
<bean id="reader1"
class="org.springframework.batch.item.xml.StaxEventItemReader">
        <property name="resource" value="users.xml" />
        <property name="fragmentRootElementName" value="user" />
        <property name="unmarshaller" ref="userUnmarshaller" />
</bean>


<bean id="userUnmarshaller"
class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
    <property name="classesToBeBound">
        <list>
            <value>com.pratap.model.User</value>
        </list>
    </property>
</bean>
```

Reading DB

- **Reading from relational databases**
  - Using JDBC item readers
    - JdbcCursorItemReader
    - StoredProcedureItemReader
    - JdbcPagingItemReader
  - Using ORM item readers
    - HibernateCursorItemReader
    - HibernatePagingItemReader
    - JpaPagingItemReader

JdbcCursor ItemReader

```
<bean id="productItemReader"
class="org.springframework.batch.item.database.JdbcCursorItemReader">
        <property name="dataSource" ref="dataSource" />
        <property name="sql"
        value="select id, name, dept, salary from user_tab" />
        <property name="rowMapper" ref="userRowMapper" />
</bean>

<bean id="userRowMapper" class="com.pratap.UserRowMapper" />
```

StoredProcedure ItemReader

```
<bean id="reader"
class="org.springframework.batch.item.database.StoredProcedureItemReader">
        <property name="dataSource" ref="dataSource" />
        <property name="procedureName" value="sp_product" />
        <property name="rowMapper" ref="productRowMapper" />
</bean>
```

Writing Data

- Spring Batch provides various implementations out of the box for writing these targets
  - Flat files
  - JSON file
  - XML files
  - Relational databases using both JDBC and ORM

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

Writing Data

- Writing files
  - Spring Batch provides item writers that write files in various formats: delimited text, fixed-field widths, and XML.

- FlatFileItemWriter
- JsonFileItemWriter
- StaxEventItemWriter

Writing flat files

- Spring Batch writes a flat file in the following steps:
  - Writes the header (optional)
  - Extracts fields for each item and aggregates them to produce a line
  - Writes the footer (optional)

- A FlatFileItemWriter uses a LineAggregator to transform each item into a String

Writing flat files

```
<bean id="productItemWriter"
class="org.springframework.batch.item.file.FlatFileItemWriter">
<property name="resource"
value="file:target/outputs/passthrough.txt" />
<property name="lineAggregator">
<bean
class="org.springframework.batch.item.file.transform.PassThroughLineAggregator" />
</property>
</bean>
```

Writing flat files

```
<bean id="productItemWriter"
class="org.springframework.batch.item.file.FlatFileItemWriter">
<property name="resource"
value="file:target/outputs/delimited-passthroughextractor.txt" />
<property name="lineAggregator">
<bean
class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
<property name="fieldExtractor">
<bean
class="org.springframework.batch.item.file.transform.PassThroughFieldExtractor" />
</property>
</bean>
</property>
</bean>
```

Writing flat files

```
<bean id="productItemWriter"
class="org.springframework.batch.item.file.FlatFileItemWriter">
<property name="resource"
value="file:target/outputs/delimitedbeanwrapperhextractor.txt" />
<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,price,name" />
</bean>
</property>
</bean>
</property>
</bean>
```

WRITING DELIMITED FILES

```
<bean id="productItemWriter"
class="org.springframework.batch.item.file.FlatFileItemWriter">
(...)
<property name="lineAggregator">
<bean
class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
<property name="delimiter" value="|">
<property name="fieldExtractor">
(...)
</property>
</bean>
</property>
</bean>
```

Writing to databases

- Spring Batch provides writers for JDBC and ORM

- Writing with JDBC
- The JdbcBatchItemWriter class is an implementation of the ItemWriter interface for JDBC that sits on top of the Spring JDBC layer, which itself hides the complexities of using the JDBC API directly.

- The main JdbcBatchItemWriter properties are sql and your choice of itemPrepared-StatementSetter or itemSqlParameterSourceProvider.

- The ItemPreparedStatementSetter class executes SQL with ? parameter markers.

- The ItemSqlParameter-SourceProvider class executes SQL statements with named parameters.

Writing to databases

```
<bean id="productItemWriter"
class="org.springframework.batch.item.database.JdbcBatchItemWriter">
<property name="assertUpdates" value="true" />
<property name="itemSqlParameterSourceProvider">
<bean
class="org.springframework.batch.item.database.BeanPropertyItemSqlParameterSourceProvider" />
</property>
<property name="sql"
value="INSERT INTO PRODUCT (ID, NAME, PRICE)
VALUES(:id, :name, :price)" />
<property name="dataSource" ref="dataSource" />
</bean>
```

Writing to databases

```
<bean id="productItemWriter"
class="org.springframework.batch.item.database.JdbcBatchItemWriter">
<property name="assertUpdates" value="true" />
<property name="itemPreparedStatementSetter">
<bean
class="com.pratap.ProductItemPreparedStatementSetter" />
</property>
<property name="sql"
value="INSERT INTO PRODUCT (ID, NAME,
PRICE)VALUES(?, ?, ?)" />
<property name="dataSource" ref="dataSource" />
</bean>
```

bulletproof jobs

- A bulletproof job is able to handle errors gracefully; it won't fail miserably because of a minor error like a missing comma.

- It won't fail abruptly, either, for a major problem like a constraint violation in the database.

bulletproof jobs

- **Robust**—The job should fail only for fatal exceptions and should recover gracefully from any nonfatal exception

- **Traceable**—The job should record any abnormal behavior. A job can skip as many incorrectly formatted lines as it wants, but it should log to record what didn't make it in the database and allow someone to do something about it.

- **Restartable**—In case of an abrupt failure, the job should be able to restart properly. Depending on the use case, the job could restart exactly where it left off or even forbid a restart because it would process the same data again.

bulletproof jobs

- Spring Batch includes three features to deal with errors: skip, retry, and restart.
  - **Skipping** allows for moving processing along to the next line in an input file if the current line is in an incorrect format. If the job doesn't process a line, perhaps you can live without it and the job can process the remaining lines in the file.
  - **Retry** attempts an operation several times: the operation can fail at first, but another attempt can succeed. Retry isn't useful for errors like badly formatted input lines; it's useful for transient errors, such as concurrency errors. Skip and retry contribute to making job executions more robust because they deal with error handling during processing.
  - **Restart** is useful after a failure, when the execution of a job crashes. Instead of starting the job from scratch, Spring Batch allows for restarting it exactly where the failed execution left off. Restarting can avoid potential corruption of the data in case of reprocessing. Restarting can also save a lot of time if the failed execution was close to the end.

Skipping instead of failing

```
<job id="importProductsJob">
<step id="importProductsStep">
<tasklet>
<chunk reader="reader" writer="writer" commit-interval="100"
skip-limit="10">
<skippable-exception-classes>
<include
class="org.springframework.batch.item.file.FlatFileParseException" />
</skippable-exception-classes>
</chunk>
</tasklet>
</step>
</job>
```

SkipPolicy

```
public class ExceptionSkipPolicy implements SkipPolicy {
private Class<? extends Exception> exceptionClassToSkip;
public ExceptionSkipPolicy(
Class<? extends Exception> exceptionClassToSkip) {
super();
this.exceptionClassToSkip = exceptionClassToSkip;
}
@Override
public boolean shouldSkip(Throwable t, int skipCount)
throws SkipLimitExceededException {
return exceptionClassToSkip.isAssignableFrom(
t.getClass()
);
}
```

Listening and logging skipped items

- Spring Batch provides the SkipListener interface to listen to skipped items:
- Once you implement the skip listener, you need to register it.
- using the listeners element in the tasklet element.

Configuring retryable exceptions

- <job id="*importProducsJob*">
- <step id="*importProductsStep*">
- <tasklet>
- <chunk reader="*reader*" *writer="writer" commit-interval="100"*
- retry-limit="*3*">
- <retryable-exception-classes>
- <include
- class="*org.springframework.dao.OptimisticLockingFailureException" />*
- </retryable-exception-classes>
- </chunk>
- </tasklet>
- </step>
- </job>

Restart on Error

- restartable <job> true / false
  - Whether the job can be restarted; default is true
- allow-start-ifcomplete <tasklet> true / false
  - Whether a step should be started even if it's already completed; default is false
- start-limit <tasklet> Integer value
  - Number of times a step can be started;is Integer.MAX_VALUE

https://www.youtube.com/watch?v=6ogkeEn1h70

res