Spring Batch POC

Table of Contents

[Introduction 3](#_Toc40955535)

[Architecture and Workflow 4](#_Toc40955536)

[Configuration 6](#_Toc40955537)

[ Maven Dependency 6](#_Toc40955538)

[ Batch Config 6](#_Toc40955539)

[ Application.properties 7](#_Toc40955540)

[ Java Configuration Sample 8](#_Toc40955541)

[ Spring Batch Job Config 9](#_Toc40955542)

[Factory Classes 11](#_Toc40955543)

[Annotations 12](#_Toc40955544)

[Listeners 13](#_Toc40955545)

[Spring Batch Steps 15](#_Toc40955546)

[Tasklet 15](#_Toc40955547)

[Uses 16](#_Toc40955548)

[Chunk 18](#_Toc40955549)

[Database Connectivity 19](#_Toc40955550)

[Reading 19](#_Toc40955551)

[Writer 22](#_Toc40955552)

[Processor 25](#_Toc40955553)

[Error Handling 27](#_Toc40955554)

[Scaling Application 29](#_Toc40955555)

[Scheduling a job 31](#_Toc40955556)

# Introduction

Spring Batch is a processing framework designed for robust execution of jobs.

Spring Batch builds upon the characteristics of the Spring while making it easy for developers to access and leverage more advance enterprise services when necessary. Spring Batch is intended to work in conjunction with a scheduler, not replace a scheduler.

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 enable extremely high-volume and high performance batch jobs though optimization and partitioning techniques. Spring Batch can be used in both simple use cases (such as reading a file into a database or running a stored procedure) as well as complex, high volume use cases (such as moving high volumes of data between databases, transforming it, and so on). High-volume batch jobs can leverage the framework in a highly scalable manner to process significant volumes of information. (Ref :<https://docs.spring.io/>)

Features

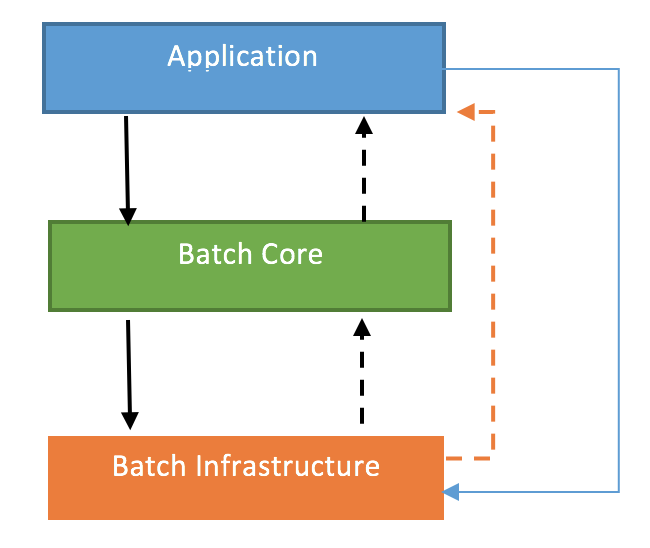
* Job Flow State Machine
* Transaction Management
* Declarative I/O
* Robust Error Handling
* Scalability

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

It also accommodates JSR-352, which is new java specification for batch processing.

# Architecture and Workflow



* **Application**: it has all the batch jobs and code
* **Batch Core**: It has all the runtime classes needed to launch and control a batch job.
* **Batch Infrastructure**: This contains the reader and writer services(ItemReader and ItemWriter) which are used by developer and the framework itself.

Spring Batch is a state machine. A Spring Batch workflow looks like this:

A close up of text on a black background

Description automatically generated

* **Job** – Defines a flow, processing which will take the program through those states
* **Step** – It represents the state in the state machine. It is an object that encapsulates sequential phase of a job and holds all the necessary information to define and control processing.
* **Job Repository –** It maintains the state of the job.
* **Job Launcher –** Creates the state of the job before launching it.

A screenshot of a social media post

Description automatically generated

* The ***ItemReader*** reads the input data and provides the found items one by one. An *ItemReader* belongs to one step and each step must have only one *ItemReader*.
* The ***ItemProcessor*** transforms items into a form that is understood by the *ItemWriter* one item at a time. An *ItemProcessor* belongs to one step and each step can have one *ItemProcessor*.
* The ***ItemWriter*** writes an information of an item to the output one item at a time. An *ItemWriter* belongs to one step and a step must have only one *ItemWriter*

(Ref: <https://www.petrikainulainen.net/programming/spring-framework/spring-batch-tutorial-introduction/>)

The input for entire step is divided into smaller chunks and that input is then read, processed and written for each chunk. For each chunk, reader will read the entire chunk, until the chunk limit is reached, and the entire chunk is then processed and passed on to ItemWriter, which will write all the items in the chunk at once. This process is repeated until all the input is exhausted.

# Configuration

## Maven Dependency

<dependency>

    <groupId>org.springframework</groupId>

    <artifactId>spring-jdbc</artifactId>

    <version>5.2.0.RELEASE</version>

</dependency>

<dependency>

    <groupId>org.springframework.batch</groupId>

    <artifactId>spring-batch-core</artifactId>

    <version>4.2.0.RELEASE</version>

</dependency>

***For Spring boot***

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-batch</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jdbc</artifactId>

</dependency>

## Batch Config

<bean id="jobRepository"

class="org.springframework.batch.core.repository.support.JobRepositoryFactoryBean">

<property name="dataSource" ref="dataSource" />

<property name="transactionManager" ref="transactionManager" />

<property name="databaseType" value="mysql" />

</bean>

<bean id="jobLauncher" class="org.springframework.batch.core.launch.support.SimpleJobLauncher">

<property name="jobRepository" ref="jobRepository" />

</bean>

<bean id="transactionManager"

class="org.springframework.batch.support.transaction.ResourcelessTransactionManager" />

<bean id="dataSource"

class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/mysql" />

<property name="username" value="root" />

<property name="password" value="" />

</bean>

<!-- Create meta-tables -->

<jdbc:initialize-database data-source="dataSource">

<jdbc:script location="org/springframework/batch/core/schema-drop-mysql.sql" />

<jdbc:script location="org/springframework/batch/core/schema-mysql.sql" />

</jdbc:initialize-database>

## Application.properties

spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/mysql

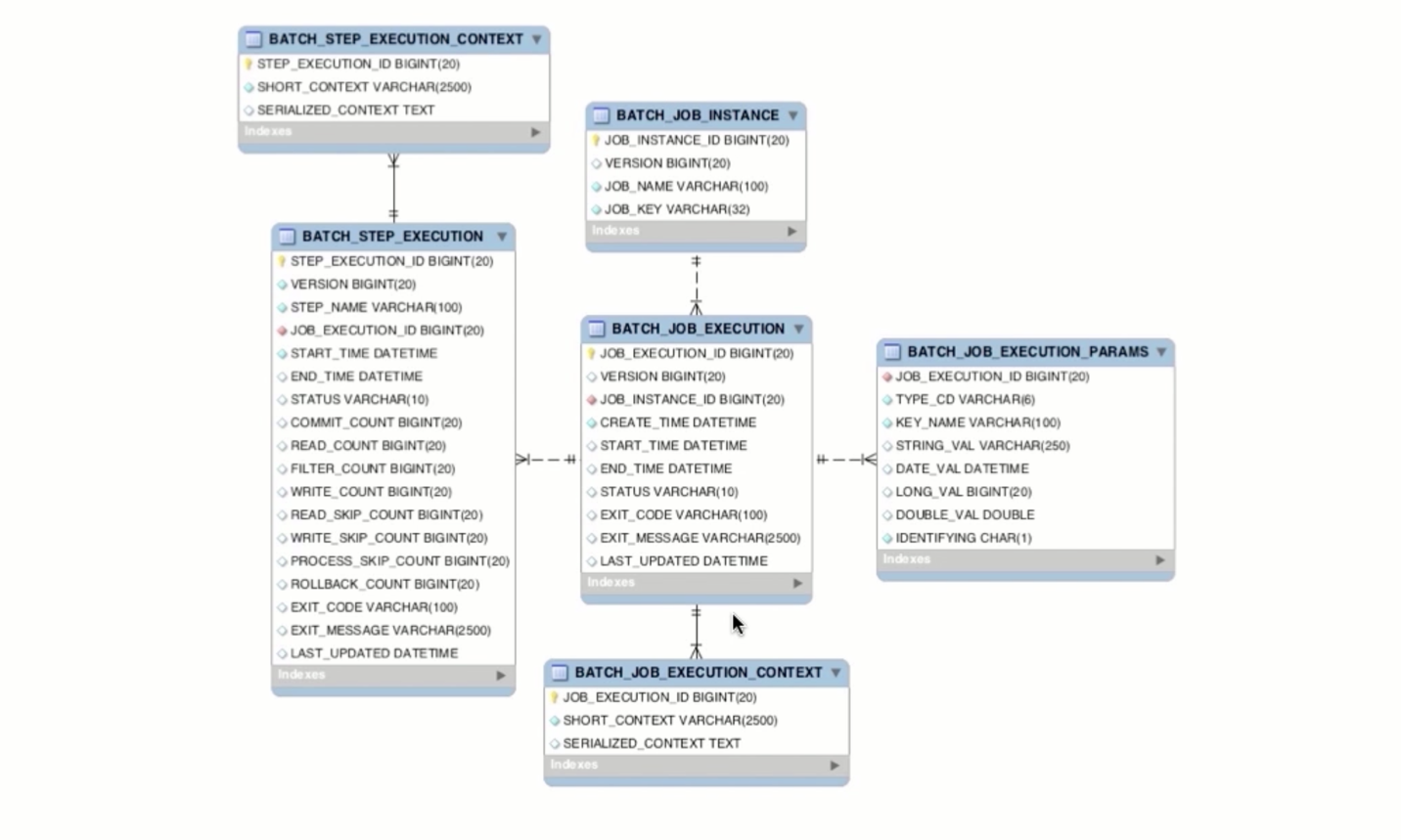
spring.datasource.username=root

spring.datasource.password=passw0rd

spring.batch.initialize-schema=ALWAYS

spring.datasource.schema=org/springframework/batch/core/schema-mysql.sql

The structure of default schema created using schema-mysql.sql looks like



## Java Configuration Sample

@Configuration

@EnableBatchProcessing

public class SpringConfig {

    @Value("org/springframework/batch/core/*schema-drop-mysql.sql*")

    private Resource dropReopsitoryTables;

    @Value("org/springframework/batch/core/*schema-mysql.sql*")

    private Resource dataReopsitorySchema;

    @Bean

    public DataSource dataSource() {

        DriverManagerDataSource dataSource = new DriverManagerDataSource();

        dataSource.setDriverClassName("*com.mysql.jdbc.Driver*");

        dataSource.setUrl("jdbc:mysql://localhost:3306/mysql");

        return dataSource;

    }

    @Bean

    public DataSourceInitializer dataSourceInitializer(DataSource dataSource)

      throws MalformedURLException {

        ResourceDatabasePopulator databasePopulator =

          new ResourceDatabasePopulator();

        databasePopulator.addScript(dropReopsitoryTables);

        databasePopulator.addScript(dataReopsitorySchema);

        databasePopulator.setIgnoreFailedDrops(true);

        DataSourceInitializer initializer = new DataSourceInitializer();

        initializer.setDataSource(dataSource);

        initializer.setDatabasePopulator(databasePopulator);

        return initializer;

    }

    private JobRepository getJobRepository() throws Exception {

        JobRepositoryFactoryBean factory = new JobRepositoryFactoryBean();

        factory.setDataSource(dataSource());

        factory.setTransactionManager(getTransactionManager());

        factory.afterPropertiesSet();

        return (JobRepository) factory.getObject();

    }

    private PlatformTransactionManager getTransactionManager() {

        return new ResourcelessTransactionManager();

    }

    public JobLauncher getJobLauncher() throws Exception {

        SimpleJobLauncher jobLauncher = new SimpleJobLauncher();

        jobLauncher.setJobRepository(getJobRepository());

        jobLauncher.afterPropertiesSet();

        return jobLauncher;

    }

}

## Spring Batch Job Config

<import resource="../config/context.xml" />

<import resource="../config/database.xml" />

<bean id="report" class="com.springBatch.model.Report" scope="prototype" />

<bean id="itemProcessor" class="com.springBatch.CustomItemProcessor" />

<batch:job id="myJob">

<batch:step id="step1">

<batch:tasklet>

<batch:chunk reader="cvsFileItemReader" writer="xmlItemWriter" processor="itemProcessor"

commit-interval="10">

</batch:chunk>

</batch:tasklet>

</batch:step>

</batch:job>

<bean id="cvsFileItemReader" class="org.springframework.batch.item.file.FlatFileItemReader">

<property name="resource" value="classpath:cvs/input/report.csv" />

<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,sales,qty,staffName,date" />

</bean>

</property>

<property name="fieldSetMapper">

<bean class="com.springBatch.ReportFieldSetMapper" />

<!-- if no data type conversion, use BeanWrapperFieldSetMapper to map by name

<bean

class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper">

<property name="prototypeBeanName" value="report" />

</bean>

-->

</property>

</bean>

</property>

</bean>

<bean id="xmlItemWriter" class="org.springframework.batch.item.xml.StaxEventItemWriter">

<property name="resource" value="file:xml/outputs/report.xml" />

<property name="marshaller" ref="reportMarshaller" />

<property name="rootTagName" value="report" />

</bean>

<bean id="reportMarshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">

<property name="classesToBeBound">

<list>

<value>com.springBatch.model.Report</value>

</list>

</property>

</bean>

# Factory Classes

* StepBuilderFactory
  + This is used for executing a step.
  + It returns a status which indicates if the particular step is Finished or it can be continued.

@Autowired

**private** StepBuilderFactory sbf;

@Bean

**public** Step step1() {

**return** sbf.get("step1").tasklet(**new** Tasklet() {

@Override

**public** RepeatStatus execute(StepContribution contribution, ChunkContext chunkContext) **throws** Exception {

System.***out***.println("Spring batch step 1 inside flow");

**return** RepeatStatus.***FINISHED***;

}

}).build();

}

* JobBuilderFactory
  + This executes a job which can compromise of a flows or steps.

@Bean

**public** Job transitionJobOnCompletion() {

System.***out***.println("Executing transition job 2");

**return** jbf.get("transitionJob2")

.start(step1())

.on("COMPLETED").to(step2())

.from(step2())

.on("COMPLETED").stopAndRestart(step3()) // stop executing in run 1. will resume from here when restarted

.from(step3())

.end()

.build();

}

# Annotations

* @Configuration
  + Spring Configuration file containing jobs, flows, steps
  + Provide a base configuration for setting up batch jobs
* @EnableBatchProcessing
  + Enable Spring Batch features
* @SpringBootApplication
  + Indicates that a class provides Spring Boot application
* @EnableAutoConfiguration
  + Attempts to automatically configure your Spring application based on the jar dependencies that you have added
* @BeforeChunk
  + Executes this before chunk is read/processed/written
* @AfterChunk
  + Executes this method after chunk has been read/processed/written
* @StepScope
  + Provide a new scope available within spring batch
  + Tells spring to instantiate the object, only once the step that is using it, calls it.

# Listeners

Listeners provided by spring batch allow a developer to take control of the execution flow at just about any junction point within the given batch job.

* JobExecutionListener
  + Provides before and after job execution listeners
* StepExecutionListener
  + Provides before and after step capabilities
* ChunckExecutionListener
  + Provides before and after chunk capabilities
  + Method to be called after an error within the chunk for extra error handling
* ItemReadListener
* ItemProcessListener
* ItemWriterListener

Sample Code

**public** **class** ChunkListener {

@BeforeChunk

**public** **void** beforeChunk(ChunkContext context) {

System.***out***.println("Before the chunk");

}

@AfterChunk

**public** **void** afterChunk(ChunkContext context) {

System.***out***.println("After the chunk");

}

}

JobExecutionListener Sample

**public** **class** JobListener **implements** JobExecutionListener {

**private** JavaMailSender jms;

/\*\*

\*

\*/

**public** JobListener(JavaMailSender sender) {

**this**.jms = sender;

}

@Override

**public** **void** beforeJob(JobExecution jobExecution) {

String jobName = jobExecution.getJobInstance().getJobName();

SimpleMailMessage mail = getSimpleMailMessage(String.*format*("%s is starting", jobName),

String.*format*("This is just to inform that %s is starting", jobName));

jms.send(mail);

}

@Override

**public** **void** afterJob(JobExecution jobExecution) {

String jobName = jobExecution.getJobInstance().getJobName();

SimpleMailMessage mail = getSimpleMailMessage(String.*format*("%s has completed", jobName),

String.*format*("This is just to inform that %s has completed", jobName));

jms.send(mail);

}

}

# Spring Batch Steps

There are two main approaches to building a step.

* Tasklet Based processing
* Chunk Oriented processing

## Tasklet

Tasklets are meant to perform a single task within a step. A Tasklet supports a simple interface that has only one method, execute(), which is called repeatedly until it either returns RepeatStatus.FINISHED or throws an exception to signal a failure. Each call to the Tasklet is wrapped in a transaction.

The below sample code is creating 2 basic steps and executing them using FlowBuilder

@Configuration

@EnableBatchProcessing

**public** **class** FlowConfiguration {

@Autowired

**private** StepBuilderFactory sbf;

@Bean

**public** Step step1() {

**return** sbf.get("step1").tasklet(**new** Tasklet() {

@Override

**public** RepeatStatus execute(StepContribution contribution, ChunkContext chunkContext) **throws** Exception {

System.***out***.println("Spring batch step 1 inside flow");

**return** RepeatStatus.***FINISHED***;

}

}).build();

}

@Bean

**public** Step step2() {

**return** sbf.get("step2").tasklet((contribution, chunkContext) -> {

System.***out***.println("Spring batch step 2 inside flow");

**return** RepeatStatus.***FINISHED***;

}).build();

}

@Bean

**public** Flow foo() {

FlowBuilder<Flow> flow = **new** FlowBuilder<Flow>("foo");

flow.start(step1())

.next(step2())

.end();

**return** flow.build();

}

}

### Uses

* Allows user to execute a series of steps
* User can pause a flow midway and the job can be resumed and continued from where it was paused

@Bean

**public** Job transitionJobOnCompletion() {

System.***out***.println("Executing transition job 2");

**return** jbf.get("transitionJob2")

.start(step1())

.on("COMPLETED").to(step2())

.from(step2())

.on("COMPLETED").stopAndRestart(step3()) // stop executing in run 1. will resume from here when restarted

.from(step3())

.end()

.build();

}

* Nested jobs can be run i.e. parent job can be configured in application.properties and the parent job will be responsible to execute the child jobs.

#Application properties

spring.batch.job.names=parentJob

@Bean

**public** Job parentJob(JobRepository jobRepository, PlatformTransactionManager transactionManager) {

Step childJobStep = **new** JobStepBuilder(**new** StepBuilder("childJobStep"))

.job(childJob)

.launcher(jobLauncher)

.repository(jobRepository)

.transactionManager(transactionManager).build();

**return** jbf.get("parentJob").start(step1Parent()).next(childJobStep).build();

}

Spring batch records 2 separate records in spring ***batch\_job\_execution*** table, one each for parent and child jobs.

* Steps can be configured to run in parallel or can be configured to run after completion of the previous step.
  + Parallel execution can be achieved by splitting the flow

@Bean

**public** Job job() {

System.***out***.println("Executing job");

**return** jbf.get("job").start(flow1())

.split(**new** SimpleAsyncTaskExecutor()).add(flow2())

.end().build();

}

* + Serial execution or flow execution can be achived like this

@Bean

**public** Job flowLastJob(Flow flow) {

System.***out***.println("Executing transition job flow last");

**return** jbf.get("flowLastJob")

.start(myLastStep())

.on("COMPLETED").to(flow)

.end()

.build();

}

## Chunk

This approach **performs actions over chunks of data**. That is, instead of reading, processing and writing all the lines at once, it'll read, process and write a fixed amount of records (chunk) at a time.

Then, it'll repeat the cycle until there's no more data in the file.

**SampleCode**

@Bean

**public** StatelessItemReader reader() {

List<String> data = **new** ArrayList<String>();

data.add("test1");

data.add("test2");

data.add("test3");

**return** **new** StatelessItemReader(data);

}

@Bean

**public** ItemWriter<String> writer() {

**return** **new** ItemWriter<String>() {

@Override

**public** **void** write(List<? **extends** String> items) **throws** Exception {

**for** (String item : items) {

System.***out***.println("Writing item " + item);

}

}

};

}

@Bean

**public** Step step1() {

**return** stepBuilderFactory.get("step12")

.<String, String>chunk(2) // chunk size

.reader(reader())

.writer(writer())

.build();

}

**Uses**

* Batch processing is more controlled. We can configure the number of records that can be read/written at once
* Better error handling when handling large data. Job resumes from where it left and the chunk already processed doesn’t need to be looked after.

# Database Connectivity

Spring batch provides various ways to connect to a database, read, process and write.

We will discuss a few ways to connect to a db or read from a csv or a xml file.

## Reading

Detailing some of the approaches.

* JdbcCursorItemReader
  + We can directly write custom sqls and execute them.

@Bean

**public** JdbcCursorItemReader<Customer> cursorItemReader() {

JdbcCursorItemReader<Customer> reader = **new** JdbcCursorItemReader<>();

reader.setSql("select id, firstName, lastName, birthdate from customer order by lastName, firstName");

reader.setDataSource(**this**.dataSource);

reader.setRowMapper(**new** CustomerRowMapper());

**return** reader;

}

* JdbcPagingItemReader
  + Uses spring batch inbuilt queryprovider to generate query

@Bean

**public** JdbcPagingItemReader<Customer> pagingItemReader() {

JdbcPagingItemReader<Customer> reader = **new** JdbcPagingItemReader<>();

reader.setDataSource(**this**.dataSource);

reader.setFetchSize(10);

reader.setRowMapper(**new** CustomerRowMapper());

MySqlPagingQueryProvider queryProvider = **new** MySqlPagingQueryProvider();

queryProvider.setSelectClause("id, firstName, lastName, birthdate");

queryProvider.setFromClause("from customer");

Map<String, Order> sortKeys = **new** HashMap<>(1);

sortKeys.put("id", Order.***ASCENDING***);

queryProvider.setSortKeys(sortKeys);

reader.setQueryProvider(queryProvider);

**return** reader;

}

* JpaPagingItemReader
* HibernatePagingItemReader
* FlatFileItemReader
  + To read from a flat file for example csv files

@Bean

**public** FlatFileItemReader<Customer> flatFileItemReader() {

FlatFileItemReader<Customer> reader = **new** FlatFileItemReader<>();

reader.setLinesToSkip(1); // skip header

reader.setResource(**new** ClassPathResource("/customer.csv"));

DefaultLineMapper<Customer> customerLineMapper = **new** DefaultLineMapper<>();

DelimitedLineTokenizer tokenizer = **new** DelimitedLineTokenizer();

tokenizer.setNames(**new** String[] {"id", "firstName", "lastName", "birthdate"});

customerLineMapper.setLineTokenizer(tokenizer);

customerLineMapper.setFieldSetMapper(**new** CustomerFieldSetMapper());

customerLineMapper.afterPropertiesSet();

reader.setLineMapper(customerLineMapper);

**return** reader;

}

* StaxEventItemReader
  + To read from a XML file.

@Bean

**public** StaxEventItemReader<Customer> customXmlItemReader() {

XStreamMarshaller unmarshaller = **new** XStreamMarshaller();

Map<String, Class> aliases = **new** HashMap<>();

aliases.put("customer", Customer.**class**);

unmarshaller.setAliases(aliases);

StaxEventItemReader<Customer> reader = **new** StaxEventItemReader<>();

reader.setResource(**new** ClassPathResource("/data/customers.xml"));

reader.setFragmentRootElementName("customer");

reader.setUnmarshaller(unmarshaller);

**return** reader;

}

* MultiResourceItemReader
  + Read from multiple files and process together
  + Useful when reading and processing records from various sources or records in split files.

@Value("classpath\*:/data/customer\*.csv")

**private** Resource[] inputFiles;

@Bean

**public** MultiResourceItemReader<Customer> multiResourceItemReader() {

MultiResourceItemReader<Customer> reader = **new** MultiResourceItemReader<>();

reader.setDelegate(multiItemReader());

reader.setResources(inputFiles);

**return** reader;

}

@Bean

**public** FlatFileItemReader<Customer> multiItemReader() {

FlatFileItemReader<Customer> reader = **new** FlatFileItemReader<>();

DefaultLineMapper<Customer> customerLineMapper = **new** DefaultLineMapper<>();

DelimitedLineTokenizer tokenizer = **new** DelimitedLineTokenizer();

tokenizer.setNames(**new** String[] {"id", "firstName", "lastName", "birthdate"});

customerLineMapper.setLineTokenizer(tokenizer);

customerLineMapper.setFieldSetMapper(**new** CustomerFieldSetMapper());

customerLineMapper.afterPropertiesSet();

reader.setLineMapper(customerLineMapper);

**return** reader;

}

## Writer

Spring provides a number of item writers to read to a file/database

* JdbcBatchItemWriter
  + Write using query

@Bean

**public** JdbcBatchItemWriter<Customer> customerItemWriter() {

JdbcBatchItemWriter<Customer> itemWriter = **new** JdbcBatchItemWriter<>();

itemWriter.setDataSource(**this**.dataSource);

itemWriter.setSql("INSERT INTO CUSTOMER VALUES (:id, :firstName, :lastName, :birthdate)");

itemWriter.setItemSqlParameterSourceProvider(**new** BeanPropertyItemSqlParameterSourceProvider());

itemWriter.afterPropertiesSet();

**return** itemWriter;

}

* JPAItemWriter
* HibernateItemWriter
* FlatFileItemWriter
  + Write to a flat file(string is stored as text).

@Bean

**public** FlatFileItemWriter<Customer> flatFileItemWriter() **throws** Exception {

FlatFileItemWriter<Customer> itemWriter = **new** FlatFileItemWriter<>();

// itemWriter.setLineAggregator(new PassThroughLineAggregator<>());

itemWriter.setLineAggregator(**new** CustomerLineAggregator());

String customerOutputPath = File.*createTempFile*("customerOutput", ".out").getAbsolutePath();

System.***out***.println(">> Output Path: " + customerOutputPath);

itemWriter.setResource(**new** FileSystemResource(customerOutputPath));

itemWriter.afterPropertiesSet();

System.***out***.println("writing new chunk");

**return** itemWriter;

}

* StaxEventItemWriter
  + Write to a XML file

@Bean

**public** StaxEventItemWriter<Customer> customXmlItemWriter() **throws** Exception {

XStreamMarshaller marshaller = **new** XStreamMarshaller();

Map<String, Class> aliases = **new** HashMap<>();

aliases.put("customer", Customer.**class**);

marshaller.setAliases(aliases);

StaxEventItemWriter<Customer> writer = **new** StaxEventItemWriter<>();

writer.setRootTagName("customers");

writer.setMarshaller(marshaller);

String customerOutputPath = File.*createTempFile*("customerOutput", ".xml").getAbsolutePath();

System.***out***.println(">> Output Path: " + customerOutputPath);

writer.setResource(**new** FileSystemResource(customerOutputPath));

writer.afterPropertiesSet();

**return** writer;

}

* CompositeItemWriter
  + Write to multiple file types.
  + They implement streams and .stream() isn’t needed

@Bean

**public** CompositeItemWriter<Customer> multipleItemWriter() **throws** Exception {

List<ItemWriter<? **super** Customer>> writers = **new** ArrayList<>();

writers.add(customXmlItemWriter());

writers.add(jsonFlatFileItemWriter());

CompositeItemWriter<Customer> cWriter = **new** CompositeItemWriter<Customer>();

cWriter.setDelegates(writers);

cWriter.afterPropertiesSet();

**return** cWriter;

}

* ClassifierCompositeItemWriter
  + Write to multiple files based on some classification

@Bean

**public** ClassifierCompositeItemWriter<Customer> classifierTtemWriter() **throws** Exception {

ClassifierCompositeItemWriter<Customer> itemWriter = **new** ClassifierCompositeItemWriter<>();

itemWriter.setClassifier(**new** CustomerClassifier(customXmlItemWriter(), jsonFlatFileItemWriter()));

**return** itemWriter;

}

@Bean

**public** Step step1() **throws** Exception {

**return** stepBuilderFactory.get("step26").<Customer, Customer>chunk(10)

.reader(pagingItemReader())

.writer(classifierTtemWriter())

.stream(customXmlItemWriter())

.stream(jsonFlatFileItemWriter())

.build();

}

## Processor

Processor acts as the business logic layer between a reader and writer. User can validate, filter, change data’s form and more before passing it to the writer.

* ItemProcessor
  + The main Item processor interface.
  + Process method is called to work upon the data

Sample – change data

@Bean

**public** ItemProcessor<Customer, Customer> processItem() {

**return** **new** ItemProcessor<Customer, Customer>() {

@Override

**public** Customer process(Customer item) **throws** Exception {

**return** **new** Customer(item.getId(), item.getFirstName().toUpperCase(), item.getLastName().toLowerCase(), item.getBirthdate());

}

};

}

Sample – filter data

@Override

**public** Customer process(Customer item) **throws** Exception {

**if** (item.getId() % 2 == 0) {

**return** **null**;

}

**return** item;

}

* ValidatingItemProcessor
  + Validates items and throws ValidationException() if validation fails
  + We can either opt to break the code or skip the failed record

@Bean

**public** ValidatingItemProcessor<Customer> validateAndProcessItem() {

ValidatingItemProcessor<Customer> processor = **new** ValidatingItemProcessor<Customer>(**new** CustomerValidator());

// instead of failing the job on exception, it will just filter out the records

processor.setFilter(**true**);

**return** processor;

}

**public** **class** CustomerValidator **implements** Validator<Customer>{

@Override

**public** **void** validate(Customer value) **throws** ValidationException {

**if** (value.getFirstName().toLowerCase().startsWith("a")) {

**throw** **new** ValidationException("Invalid name. Should not start with 'a' : " + value.getFirstName());

}

}

}

* CompositeItemProcessor
  + Allows user to process the records in multiple different ways.
  + Can be handy to modularize the code and keeping the validation block separate from filtering and processing blocks

@Bean

**public** CompositeItemProcessor<Customer, Customer> compositeItemProcessor() **throws** Exception{

List<ItemProcessor<Customer,Customer>> processors = **new** ArrayList<>();

processors.add(filterItemProcessor());

processors.add(processItem());

CompositeItemProcessor<Customer, Customer> processor = **new** CompositeItemProcessor<Customer, Customer>();

processor.setDelegates(processors);

processor.afterPropertiesSet();

**return** processor;

}

# Error Handling

Error Handling can be divided into 3 parts – Restart, Retry, Skip

* Restart
  + In case the job fails due to some reason, logs are recorded. The next instance of job, after restart, will be from the step it failed

@Bean

@StepScope

**public** Tasklet restartTasklet() {

**return** **new** Tasklet() {

@Override

**public** RepeatStatus execute(StepContribution contribution, ChunkContext chunkContext) **throws** Exception {

Map<String, Object> stepExecutionContext = chunkContext.getStepContext().getStepExecutionContext();

**if**(stepExecutionContext.containsKey("ran")) {

System.***out***.println("Step Executed 2nd time. Success");

**return** RepeatStatus.***FINISHED***;

}

**else** {

System.***out***.println("Fail the step in 1rst go");

chunkContext.getStepContext().getStepExecution().getExecutionContext().put("ran", **true**);

**throw** **new** RuntimeException("step failed");

}

}

};

}

* Retry
  + The job can be setup in a such a way that for certain type of exceptions, it will automatically retry, a pre configured number of times, if it fails
  + Can be useful in case of connection failures

@Bean

**public** Step step1() {

**return** stepBuilderFactory.get("step31")

.<String, String>chunk(10)

.reader(reader())

.processor(processor(**null**))

.writer(writer(**null**))

.faultTolerant()

.retry(CustomRetryableException.**class**)

.retryLimit(15)

.build();

}

* Skip
  + In case of exception, the item can also be skipped and the rest of the records can be processed.
  + This can be useful when processing huge chunk of independent data
  + Chunk size is reduced to 1 while writing, for the items in current chunk, the moment an exception occurs and skip logic is triggered
  + Skip Listener can also be defined which can help in logging the skipped record

@Bean

**public** Step step1() {

**return** stepBuilderFactory.get("step")

.<String, String>chunk(10)

.reader(reader())

.processor(processor())

.writer(writer())

.faultTolerant()

.skip(CustomException.**class**)

.skipLimit(15)

.listener(**new** CustomSkipListener())

.build();

}

**public** **class** CustomSkipListener **implements** SkipListener {

@Override

**public** **void** onSkipInRead(Throwable t) {

}

@Override

**public** **void** onSkipInWrite(Object item, Throwable t) {

System.***out***.println(">> Skipping " + item + " because writing it caused the error: " + t.getMessage());

}

@Override

**public** **void** onSkipInProcess(Object item, Throwable t) {

System.***out***.println(">> Skipping " + item + " because processing it caused the error: " + t.getMessage());

}

}

# Scaling Application

* Multi threaded step
  + Each chunk can be processed parallel in its own thread
    - SympleAsyncTaskExecutor – This creates a new thread everytime a task is executed. Not for production environment.
    - ThreadPoolTaskExecutor – Recommended for production environment
  + Restartability will not possible as ability to maintain state is lost

@Bean

**public** Step step1() **throws** Exception {

**return** stepBuilderFactory.get("step1")

.<Customer, Customer>chunk(1000)

.reader(pagingItemReader())

.writer(customerItemWriter())

.taskExecutor(**new** SimpleAsyncTaskExecutor())

.build();

}

* AsyncItemProcessor and AsyncItemWriter
  + Item processor return a future
  + Actual processor logic is executed via a thread
  + When logic is returned from the thread, future is returned and item writer unwraps the future and writes the data

@Bean

**public** AsyncItemProcessor asyncItemProcessor() **throws** Exception {

AsyncItemProcessor<Customer, Customer> asyncItemProcessor = **new** AsyncItemProcessor();

asyncItemProcessor.setDelegate(itemProcessor());

asyncItemProcessor.setTaskExecutor(**new** SimpleAsyncTaskExecutor());

asyncItemProcessor.afterPropertiesSet();

**return** asyncItemProcessor;

}

@Bean

**public** AsyncItemWriter asyncItemWriter() **throws** Exception {

AsyncItemWriter<Customer> asyncItemWriter = **new** AsyncItemWriter<>();

asyncItemWriter.setDelegate(customerItemWriter());

asyncItemWriter.afterPropertiesSet();

**return** asyncItemWriter;

}

* Partitioning
  + Divide the data into partition
    - Via threads
    - Remotely
* Remote Chunking
  + Read is done by the master, Processing and writing occurs in a slave

# Scheduling a job

Spring batch doesn’t provide a out of the box scheduler but spring scheduler can itself be used for the purpose