Spring Batch

Jorge Simão, Ph.D.

Spring Batch Overview

Spring Batch is Java batch framework providing abstractions, components and services useful to build reliable batch applications. Batch applications are characterized by processing of large amounts of data in a long-running and repeated way, thus have very specific requirements for reliability.

Spring Batch infrastructure and components are able to keep persistent track of work progress allowing jobs to be restart from the point they were stopped in case of failures. This is specially useful, since the the larger the dataset the higher the probability of failure.

Spring Batch relies on **Core Spring Framework** dependency injection for job definition and infrastructure configuration. In addition to the core infrastructure components, **Spring Batch** comes also with a large number of out-of-the-box components that can be used to easily write batch application (e.g. to read and write data from files and databases).

A companion project **Spring Batch Admin**, provides a web GUI for launching jobs and inspecting job status interactively. **Spring Batch** is also the reference implementation for **Java Batch** JSR-352 standard API.

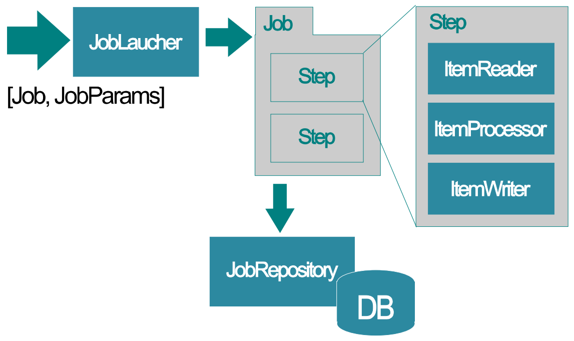
» Example: Importing Batch Core Dependency [Maven]

```
<dependency>
  <groupId>org.springframework.batch</groupId>
  <artifactId>spring-batch-core</artifactId>
  <version>3.0.7.RELEASE</version>
</dependency>
```

Spring Batch Concepts

A **Job** is defined as a computation or activity for data processing. A **Job** is composed of one or more **Step**, with each **Step** defining a stage in the overall computation. A **Step** can perform any arbitrary computation (e.g. by calling a method in a Spring bean). A common type of **Step** is a *chunk-based* **Step**, which processes data items iteratively with a sequence of *read-process-write* actions.

A **JobInstance** is defined as a **Job** with a set of *key--value* parameters modelled with **JobParams**. Parameters define the data to be processed (e.g. a file name, a queue name in a message-broker, or table name or ID range). **JobLauncher** is the engine used to control the creation and execution of each **JobInstance**. A **JobInstance** is created and started by submitting the **Job** and **JobParams** to the **JobLauncher**.



Internally, a **JobExecution** is created for each attempt to complete the execution of a **JobInstance**. And for each **Step** in a **Job**, a **StepExecution** is created as part of a owning **JobExecution**. Meta-data about all these data-structures is persisted via a **JobRepository**

Batch Infrastructure

Spring Batch infrastructure setup requires the definition of a **JobLauncher** and a **JobRepository** as Spring managed singleton beans. The default implementation for a **JobLauncher** is a **SimpleJobLauncher**. It runs **Jobs** synchronously by default, but can be configured with a **TaskExecuter** to run **Jobs** asynchronously.

An out-of-the-box implementation of a **JobRepository** is available that persists **JobExecution** meta-data in a relational database. It can be configured in XML with element **<job-repository>** from **<batch:*>** namespace. Property **data-source** specifies the **DataSource** to use. The **JobRepository** tables are created with prefix **BATCH_**, but this can be changed with attribute **table-prefix**.

» Example: Batch Infrastructure Configuration [XML]

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:batch="http://www.springframework.org/schema/batch"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="...">

  <batch:job-repository id="jobRepository"
    data-source="dataSource"/>

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

» Example: JobLaucher with TaskExecutor [XML]

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

<task:executor id="taskExecutor" />
```

Defining Jobs & Steps

A **Job** is defined in XML with element **<job>**, together with a set of **Step** defined with element **<step>**. The computation performed by a **Step** is specified with element **<tasklet>**. For arbitrary kinds of steps (e.g. move a file, or delete a directory content) a reference to a bean implementing interface **Tasklet** is provided in attribute **ref**. A POJO can also be used by setting attribute **method**.

For a *chunk-based* **Step**, the element **<chunk>** should be additionally specified. Attributes **reader**, **processor**, **writer**, specify, respectively, a **ItemReader** – a strategy to read items and map then to domain objects, **ItemProcessor** – to process individual items, and a **ItemWriter** – to write bundled items (a chunk). Out-of-the-box implementation of **ItemReader** and **ItemWriter** can be used in most cases, while custom implementations of **ItemProcessor** are required whenever any kind of item processing is required. Attribute **commit-interval** specifies the chunk size. Transactions boundaries match the chunk size (i.e. transaction commit is done after the call to the **ItemWriter**).

A component used in the configuration of a **Step** whose settings include SPEL expressions should be set of scope **step** to enforce late evaluation of the expression and late creation of the component (e.g. **#{jobParams[...]}**).

» Example: Job with Chunk-Based Step

```
<batch:job id="job">
  <batch:step id="step">
    <batch:tasklet>
      <batch:chunk reader="itemReader" writer="itemWriter"
processor="itemProcessor" commit-interval="100"/>
    </batch:tasklet>
  </batch:step>
</batch:job>
```

» Example: Job with Custom Tasklet

```
<batch:job id="job">
    <batch:step id="step">
       <batch:tasklet ref="mystep" method="run" />
    </batch:step>
</batch:job>

<bean id="mytasklet" class="myapp.MyTasklet" />

public class MyTasklet implements Tasklet {
  @Override
  public RepeatStatus execute(StepContribution step,
ChunkContext context) throws Exception {
    //....
    return RepeatStatus.FINISHED;
  }
}
```

In **Jobs** with multiple **Step**, steps are most commonly executed sequentially. The attribute **next** defines the ordering of **Step**. More complex ordering are possible. Element **<next>** defines the transition to occur when a step completes with a certain **ExitStatus**. Elements **<end>**, **<fail>**, and **<stop>** can also be used to terminated a **Step** with status of **COMPLETED**, **FAILED**, and **STOPED**. Element **<decision>** allows a **JobExecutionDecider** to determine the **ExitStatus** of a **Step**, and express more complex **Step** orderings.

» Example: Multi-Step Job

```
<batch:job id="job">
  <batch:step id="step" next="step2">
    <batch:tasklet ref="setup"/>
  </batch:step>
  <batch:step id="step2" >
    <batch:tasklet ref="copy"/>
  </batch:step>
</batch:job>
```

Running Jobs

A **Job** is run by calling method **JobLaucher.run()** with a reference to the **Job**. Instances of the same **Job** are distinguished with a unique collection of parameters represented with **JobParams**. The fluent API of **JobParamsBuilder** can be used to created a **JobParams**. Submitting a **Job** with the same parameters make the **Job** instance to restart its execution – if it previously stopped. If a **JobInstance** was completed with success and it is submitted again, the exception **JobInstanceAlreadyCompleteException** is thrown. **JobLaucher.run()** returns an instance of **JobExecution** which can be used to check status of the execution.

» Example: Submitting Job Instance to JobLauncher

```
@Autowired
JobLauncher jobLauncher;
```

```
@Autowired @Qualifier("job1")
Job job;

public void runJob() throws Exception {
  JobParameters params = new JobParametersBuilder()
    .addString("resource", "data.csv").toJobParameters();
  JobExecution jobExecution = jobLauncher.run(job, params);
  System.out.println(jobExecution.getExitStatus() + " " +
                     jobExecution.getStatus());
}
```

A **Job** can also be run using the provided command-line tool **CommandLineJobRunner**. It take as parameters: a filename for a XML spring bean file containing the **Job** definition(s) and Batch infrastructure; the name of the **Job** to run; and key-value pairs used as **Job** parameters.

» Example: Running a Job w/ CommandLineJobRunner

```
$ java CommandLineJobRunner endOfDayJob.xml endOfDay
schedule.date(date)=2007/05/05
```

FlatFile ItemReader & ItemWriter

FlatFileItemReader reads items from *flat files* – text files containing a 2D collection of records (e.g. CSV). Property **resource** specifies the file to read. Property **lineMapper** specifies a strategy to map a line of text to a record. The **DefaultLineMapper** implementation perform this in two steps: a **LineTokenizer** maps a line of text to a **FieldSet** – a generic representation for a record with *key–value* pairs; and a **FieldSetMapper** that maps a **FieldSet** to the domain object returned by the **ItemReader**. For CSV like file, a **DelimitedLineTokenizer** is used as implementation for the **LineTokenizer**. The default field separator is the **','**, but this can changed. A useful out-of-the-box implementation of a **FieldSetMapper** is the **BeanWrapperFieldSetMapper** that maps fields in a **FieldSet** to Java bean properties of a specified target type. For cases where the **ItemProcessor** or **ItemWritter** use directly a **FieldSet**, the implementation **PassThroughFieldSetMapper** is available.

» Example: FlatFileItemReader for Reading CSV Files

```
<bean id="itemReader" class="org.springframework.batch
   .item.file.FlatFileItemReader">
  <property name="resource" value="classpath:data.csv"/>
  <property name="lineMapper">
    <bean class="org.sfpringframework.batch
       .item.file.mapping.DefaultLineMapper">
      <property name="lineTokenizer">
        <bean class="org.springframework.batch
          .item.file.transform.DelimitedLineTokenizer">
            <property name="names" value="name,score" />
        </bean>
      </property>
      <property name="fieldSetMapper">
        <bean class="org.springframework.batch
          .item.file.mapping.PassThroughFieldSetMapper"/>
      </property>
    </bean>
  </property>
</bean>
```

» Example: FlatFileItemReader w/ FixedLengthTokenizer

```
<bean id="itemReader" class="org.springframework.batch
  .item.file.FlatFileItemReader">
  <property name="resource" value="classpath:products.csv"/>
  <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="names" value="Code,Price,Stock" />
          <property name="columns" value="1-10,11-20,21-30" />
        </bean>
      </property>
     <property name="fieldSetMapper">
        <bean class="org.springframework.batch
            .item.file.mapping.BeanWrapperFieldSetMapper">
           <property name="targetType" value="myapp.Product"/>
        </bean>
      </property>
    </bean>
  </property>
</bean>
```

» Example: Custom FieldSetMapper

```
public class ProductFieldSetMapper implements
FieldSetMapper<Product> {
  @Override
  public Product mapFieldSet(FieldSet fs)
  throws BindException {
    return new Product(fs.readString("Code"),
          fs.readDouble("Price"),  fs.readDouble("Stock"));
  }
}
```

A **FlatFileItemWriter** can be use to write to flat files. The property **lineAggregator** strategy is used to map a domain object into a text line. The out-of-the-box implementations of **LineAggregator** use a strategy **FieldExtractor** to map first a domain object to a **Object[]**, which is then mapped to a text line. **DelimitedLineAggregator** maps the **Object[]** to a comma-separated fields. **FormatterLineAggregator** outputs a formatted line. For items of type **FieldSet** or a **Collection** the **PassThroughFieldExtractor** can be used as **FieldExtractor**.

» Example: FlatFileItemWriter outputting CSV File

```
<bean id="itemWriter" class="org.springframework.batch
  .item.file.FlatFileItemWriter">
    <property name="resource" value="file:target/output.csv" />
    <property name="lineAggregator">
       <bean
class="org.springframework.batch.item.file.transform.DelimitedL
ineAggregator">
         <property name="delimiter" value=","/>
         <property name="fieldExtractor">
            <bean class="org.springframework.batch
             .item.file.transform.PassThroughFieldExtractor" />
         </property>
       </bean>
    </property>
</bean>
```

XML ItemReader & ItemWriter

XML files is another commonly used format for data import/export – where each item is represented by a XML fragment. Class **StaxEventItemReader** is configured with an **Unmarshaller** to map each XML fragment to an item. Conversely, class **StaxEventItemWriter** is configured with a **Marshaller** to map each item into an XML fragment.

» Example: Reading Items from XML File

```
<bean id="xitemReader" class="org.springframework.batch
  .item.xml.StaxEventItemReader">
  <property name="fragmentRootElementName" value="product" />
  <property name="resource" value="classpath:products.xml" />
  <property name="unmarshaller" ref="marshaller" />
</bean>

<bean id="marshaller" class="org.springframework.oxm
  .xstream.XStreamMarshaller">
  <property name="aliases">
    <util:map id="aliases">
      <entry key="product" value="myapp.domain.Product" />
    </util:map>
  </property>
</bean>
```

JDBC ItemReader & ItemWriter

Reading from relational databases with JDBC can be done with a **JdbcCursorItemReader**. A SQL query is executed and a **ResultSet** is iterated for data streaming. Property **dataSource** specifies the **DataSource** to read the data from, property **sql** specifies the SQL query, and property **rowMapper** specifies instance of a **RowMapper** as a strategy to map a row to a domain object. (The same abstraction used in **JdbcTemplate** from **Core Spring Framework**.)

JdbcPagingItemReader provides an alternative strategy to read from a database with JDBC using pagination – i.e. a query is executed multiple times with different values in the *limit clause*. Property **pageSize** specifies the page size. A **QueryProvider** strategy is used to generates the complete SQL query, and to accommodate the fact that different databases use different SQL syntax for the limit clause. **SqlPagingQueryProviderFactoryBean** detects the type of **dataSource** being used, and automatically selects a suitable **QueryProvider**. The *select* and *from* clause, and optionally the *where* clause, for the SQL query need to be specified as separated properties. Property **sortKey** is also required to specify the column for ordering.

» Example: JdbcCursorItemReader

```
<bean id="itemReader" class="org.springframework.batch
  .item.database.JdbcCursorItemReader">
  <property name="dataSource" ref="dataSource"/>
  <property name="sql" value="select * from products"/>
  <property name="rowMapper" ref="productMapper"/>
</bean>

<bean id="productMapper" class="myapp.ProductMapper" />
```

» Example: Reading Database w/ Pagination

```
<bean id="itemReader4"
class="org.sf.batch.item.database.JdbcPagingItemReader">
  <property name="dataSource" ref="dataSource"/>
  <property name="pageSize" value="100"/>
  <property name="rowMapper" ref="productMapper"/>
  <property name="queryProvider">
    <bean class="org.sf.batch
    .item.database.support.SqlPagingQueryProviderFactoryBean">
      <property name="selectClause" value="select Code, Price,
Stock, Sales"/>
      <property name="fromClause" value="from product"/>
      <property name="whereClause" value="where stock>0"/>
      <property name="sortKey" value="id"/>
     </bean>
  </property>
</bean>
```

Writing to a relational database with JDBC can be done with the out-of-the-box implementation **JdbcBatchItemWriter**. It creates a **PreparedStatement** and uses it for a batch insert/write. The strategy **ItemPreparedStatementSetter** is used to set the columns values from an item on the **PreparedStatement**.

» Example: Batch Writing Items w/ JDBC

```
<bean id="itemWriter" class="org.springframework.batch
  .item.database.JdbcBatchItemWriter">
  <property name="dataSource" ref="dataSource"></property>
  <property name="sql" value="insert into product values
(?,?,?)"/>
  <property name="jdbcTemplate" ref="jdbcTemplate"/>
  <property name="itemPreparedStatementSetter"
ref="productPreparedStatementSetter"/>
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core
  .namedparam.NamedParameterJdbcTemplate">
  <constructor-arg ref="dataSource"/>
</bean>

<bean id="productPreparedStatementSetter"
class="myapp.ProductPreparedStatementSetter" />
```

» Example: A Custom PreparedStatementSetter

```
public class ProductPreparedStatementSetter implements
ItemPreparedStatementSetter<Product> {
  @Override
  public void setValues(Product product, PreparedStatement ps)
throws SQLException {
    ps.setString(1, product.getCode());
    ps.setDouble(2, product.getPrice());
    ps.setDouble(3, product.getStock());
  }
}
```

Custom ItemReader & ItemWriters

A custom **ItemReader** is defined my implementing method

read() – which returns the next item to process and/or write (usually a domain object, or a **FieldSet**), or **null** if no more items are available. A **ItemReader** can be made reliable and stateful by using a **ExecutionContext** – an abstraction with a **Map** like key-value API, whose values are saved and recovered persistently. Interface **ItemStream** defines callbacks to access and update a **ExecutionContext** contextualized on a **StepExecution** to can be used to keep track of **Step** progress in some custom variable (e.g. number of processed lines, or ID cursor for current row).

A custom **ItemWriter** is defined by implementing method **write()** to write a collection of items (chunk). Batch write operations are a recommended approach to increase performance.

A **ItemProcessor** is defined by implementing method **process()** to code some custom processing (e.g. item transformation or mapping). The input and output types of items may be different. Returning **null** makes an item to be ignored (i.e. is not written).

» Example: A Custom ItemProcessor

```
public class ProductProcessor implements ItemProcessor<Product,
Product>{
  @Override
  public Product process(Product product) throws Exception {
    product.setSales(0);
    return product.getStock()>0 ? product : null;
  }
}
```

Job Recovery

Spring Batch uses several mechanisms to increase the reliability of batch processing. Chunk-based steps can be configured with a *retry* and *skip* policy to deal with errors during item reading/writing/processing. A retry should be attempted if the exception throw while processing an item is potentially temporary. The attribute **retry-limit** in element **<chunk>** specifies the maximum number of times a item is attempted to be processed in case of errors. Sub-element **<retryable-exception-classes>** limits the set of exceptions considered temporary. For more complex use cases a **RetryPolicy** strategy can be specified as bean reference in attribute **retry-policy**. Applications can be informed about retried items my implementing interfaces **ItemReadListener**. **ItemProcessListener**, or **ItemWriteListener** and registering it inside element **<listeners>**.

An item should be skipped when failure to perform the processing should not lead to an immediate **Step**/**Job** abort. The attribute **skip-limit** specifies the maximum number of items that can be skipped before aborting a **Step**/**Job**. Sub-element **<skippable-exception-classes>** limits the set of exceptions considered skippable. For more complex use cases, a **SkipPolicy** strategy can be specified as bean reference in attribute **skip-policy**. Applications can keep track of skipped items by implementing a **SkipListener** and registering it inside element **<listeners>**.

» Example: Step with Retry & Skip Configuration

```
<batch:job id="job" >
  <batch:step id="step" >
    <batch:tasklet start-limit="3">
      <batch:chunk reader="itemReader" writer="itemWriter"
processor="itemProcessor" commit-interval="100" retry-limit="5"
skip-limit="10" >
        <batch:retryable-exception-classes>
          <batch:include class="org.sfpringframework
   .dao.DeadLockLoserDataAccessException"/>
        </batch:retryable-exception-classes>
        <batch:skippable-exception-classes>
          <batch:include class="org.springframework.batch
                 .item.file.FlatFileParseException"/>
        </batch:skippable-exception-classes>
        <batch:listeners>
          <batch:listener><bean class="myapp.MyStepListener"/>
          </batch:listener>
        </batch:listeners>
      </batch:chunk>
    </batch:tasklet>
  </batch:step>
</batch:job>
```

By default, a **Step** may be restarted as many times as needed to complete. The attribute **start-limit** in element **<tasklet>** sets a limit on the maximum number of attempts allowed to run a **Step**. Conversely, once a **Step** completes with success it can no longer be started again. The attribute **allow-start-if-complete="true"** modifies this behavior to always run a **Step** even if it run with success previously (e.g. to always perform a clean up or preparation step for following **Step** in the same **Job**).

» Example: Setting Restart Policy for Steps

```
<batch:job id="job">
  <batch:step id="step" next="step2">
    <batch:tasklet allow-start-if-complete="true" ref="setup"/>
  </batch:step>
  <batch:step id="step2" >
    <batch:tasklet start-limit="3"  >
      <batch:chunk reader="itemReader" writer="itemWriter"
processor="itemProcessor" commit-interval="100"/>
    </batch:tasklet>
  </batch:step>
</batch:job>
```

Job Paralelization

When jobs have to process large amounts of date and take too long to run to completion, it is useful to parallelize the execution of long-running **Step**. **Spring Batch** supports several strategies for parallelization. A **Step** can be executed concurrently by multiple threads by setting attribute **task-executor** with a reference to a bean of type **TaskExecutor** –

e.g. defined using the **<task:executor>** element. In a chunk-based step, each chunk is executed in a separated task/thread. Attribute **throttle-limit** specifies the number of concurrent tasks to use (4 is the default). A limitation of this approach, is that most out-of-the-box **ItemReader** in the framework are stateful and not thread-safe, so they can not be used directly. Order of items in the output, also may also deviate from the order in the input.

For **Jobs** that contain multiple logically independent step, an approach to parallel execution is to have each **Step** executed as a concurrent flow. Element **<split>** defines a splited **Step**, with multiple independent **Step** (or sequence) defined as a **<flow>**.

» Example: Multi-Thread Step

```
<batch:job id="job1" >
  <batch:step id="step1" >
    <batch:tasklet start-limit="3"  task-
executor="taskExecutor" throttle-limit="10">
      <batch:chunk reader="itemReader" writer="itemWriter"
processor="itemProcessor" commit-interval="100"/>
    </batch:tasklet>
  </batch:step>
</batch:job>

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

» Example: A Splitted Step w/ Multiple Flows

```
<batch:job id="job4" >
  <batch:split id="split1" next="step4">
    <batch:flow>
      <batch:step id="step1" next="step2">
        <batch:tasklet ref="prepare" />
      </batch:step>
      <batch:step id="step2">
        <batch:tasklet> ... </batch:tasklet>
      </batch:step>
    </batch:flow>
    <batch:flow>
      <batch:step id="step3">
        <batch:tasklet> ... </batch:tasklet>
      </batch:step>
    </batch:flow>
</batch:split>
<batch:step id="step4" >
  <batch:tasklet ref="cleanup" />
</batch:step>
</batch:job>
```

An alternative approach to **Step** parelization is to statically *partition* the data to be processed and process each part as a separated slave **Step**. The element **<partition>** defines a partitioned step. The interface **Partitioner** defines a partition strategy, that returns a **Map** containing the **ExecutionContext** for each individual slave **Step**. Commonly, one or more parameters are used to defined a partition – i.e. the items that each slave **Step** should process (e.g. filename or ID range). The **ItemReader** should be configured to use the partition parameters in a SPEL expression to select which items to process: **#{stepExecutionContext[...]}**.

The attribute **handler** defines an instance **PartitionHandler** responsible for the execution of the slaves. The element **<handler>** configures an out-of-the-box implementation **TaskExecutorPartitionHandler** that executes each step in a separated task submitted to a **TaskExcecutor**.

» Example: Partitioned Step

```
<batch:job id="job5" >
  <batch:step id="step1.master" >
    <batch:partition step="step-1" partitioner="mypartitioner">
       <batch:handler task-executor="taskExecutor" grid-
size="10" />
    </batch:partition>
  </batch:step>
</batch:job>

<batch:step id="step-1">
  <batch:tasklet>
    <batch:chunk reader="itemReader" writer="itemWriter"
processor="itemProcessor" commit-interval="100"/>
  </batch:tasklet>
</batch:step>

<bean id="itemReader8" class="org.springframework.batch
  .item.file.FlatFileItemReader">
  <property name="resource" value=
    "#{stepExecutionContext[resource.name]}/*"/>
  <property name="lineMapper"> … </property>
</bean>

<bean id="mypartitioner" class="myapp.MyPartitioner"/>
```

» Example: Custom Partitioner By ID Range

```
public class MyPartitioner implements Partitioner {
  @Override
  public Map<String, ExecutionContext> partition(int gridSize){
    Map<String, ExecutionContext> map = new HashMap<String,
ExecutionContext>();
    int maxId = 1000; //query DB
    int id = 1;
    for (int i = 1; i <= gridSize; i++) {
      ExecutionContext context = new ExecutionContext();
      context.putInt("id0", id);
      id += maxId/gridSize;
      context.putInt("id1", id);
      map.put("partition" + i, context);
    }
    return map;
  }
}
```

Java Batch DSL

Spring Batch support Java-based configuration of **Jobs** and **Steps** (since 2.2.0). Annotation **@EnableBatchProcessing** auto-configures two builders – **JobBuilderFactory** and **StepBuildFactory**, and the batch infrastructures beans including **JobRepository** and **JobLauncher**. **JobBuilderFactory** provides a fluent API (DSL) to define **Jobs** in Java, while **StepBuildFactory** provides a fluent API to define **Steps**.

» Example: Job Definition w/ Java DSL

```
@Configuration
@EnableBatchProcessing
public class BatchConfig {
  @Autowired private JobBuilderFactory jobs;

  @Autowired private StepBuilderFactory steps;

  @Bean public Job job(@Qualifier("step1") Step step1,
                       @Qualifier("step2") Step step2) {
    return jobs.get("myJob").start(step1)
             .next(step2).build();
  }

  @Bean protected Step step1(ItemReader<Product> reader,
    ItemProcessor<Product, Product> processor,
    ItemWriter<Product> writer) {
      return steps.get("step1")
          .<Product, Product> chunk(100)
          .reader(reader)
          .processor(processor)
          .writer(writer).build();
    }

  @Bean protected Step step2(Tasklet tasklet) {
    return steps.get("step2")
    .tasklet(tasklet).build();
  }

  @Bean protected Tasklet tasklet() {
    return new MyTasklet();
  }
}
```

Spring Batch w/ Spring Boot

Spring Boot includes auto-configuration and auto-execution support for **Spring Batch**. Boot support is enabled with Maven dependency **spring-boot-starter-batch**. This dependency implies automatic auto-configuration of **Spring Batch** infrastructure beans – making annotation **@EnableBatchProcessing** optional. **Spring Boot** also runs automatically all **Job**s defined in the **ApplicationContext**, or the **Job**s whose names are specified in environment variable **spring.batch.job.names**. If a **JobRegistry** bean is defined, the **Job**s set in variable **spring.batch.job.names** are resolved from the **JobRegistry**, rather than taken as bean names. This is useful in applications where **Job**s are defined across multiple **ApplicationContext** and registered centrally in the **JobRegistry**.

» Example: Importing Batch Boot Starter [Maven]

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-batch</artifactId>
</dependency>
```

Resources

- Spring Batch Project home page: http://projects.spring.io/spring-batch/
- Spring Batch Reference Manual: http://docs.spring.io/spring-batch/4.0.x/reference/html/index.html
- Spring Batch Admin: http://docs.spring.io/spring-batch-admin/

About the Author

Jorge Simão is a software engineer and IT Trainer, with two decades long experience on education delivery both in academia and industry. Expert in a wide range of computing topics, he his an author, trainer, and director (Education & Consulting) at **EInnovator**. He holds a B.Sc., M.Sc., and Ph.D. in Computer Science and Engineering.

Spring Integration Training

Enterprise Spring is a 4-day trainer lead course that teaches how to use **Spring Framework**, **Spring Integration**, **Spring Batch**, and **Spring XD** to build enterprise integration solutions. Completion of this training prepares participants to take a certification exam and become a *Pivotal Certified* **Enterprise Integration Specialist**.
Book for a training event in a date&location of your choice: **www.einnovator.org/course/enterprise-spring**