VERSION 0.1

**Mridul Mohit, Technology Services**

**UST Global**

Spring boot Batch

Basic Introduction Session

Table of Contents

[1 Spring Boot Batch Introduction 1](#_Toc165871244)

[2 Basic Project Set up and Requirements 1](#_Toc1788825067)

[3 Basic Annotations and classes 2](#_Toc235845356)

[4 Spring Boot Batch Code Components 8](#_Toc143255691)

[5 Application Properties profiles, Database set up 23](#_Toc1947447967)

[6 Spring Boot Batch – Deployment, Scheduling 25](#_Toc1741589382)

[7 Spring Boot Batch – Spring framework 6 upgrade 27](#_Toc1017686145)

[8 Spring Boot Batch – OOB Tables 28](#_Toc130686753)

[9 Parallel Processing- Partition and Flow 28](#_Toc1932244072)

[10 References 28](#_Toc2049440979)

# Spring Boot Batch Introduction

**Batch Processing**: -

1. Batch Processing allows businesses to process large volumes of data quickly, resulting in faster and more efficient Business Intelligence. Many records can be processed at once in batch processing, which reduces processing time and ensures that data is delivered on time.
2. Furthermore, because multiple jobs can be handled concurrently, it makes business intelligence faster than ever before.
3. Batch processing reduces the likelihood of errors by automating most or all components of a processing job and minimizing user interaction.
4. Earlier most of the batch development was done by creating a from scratch Core Java Batch Standalone application where developer must take care of the threading, batch transactions, resource loading etc. With the introduction of Spring Boot Batch framework batch processing has become very easy.

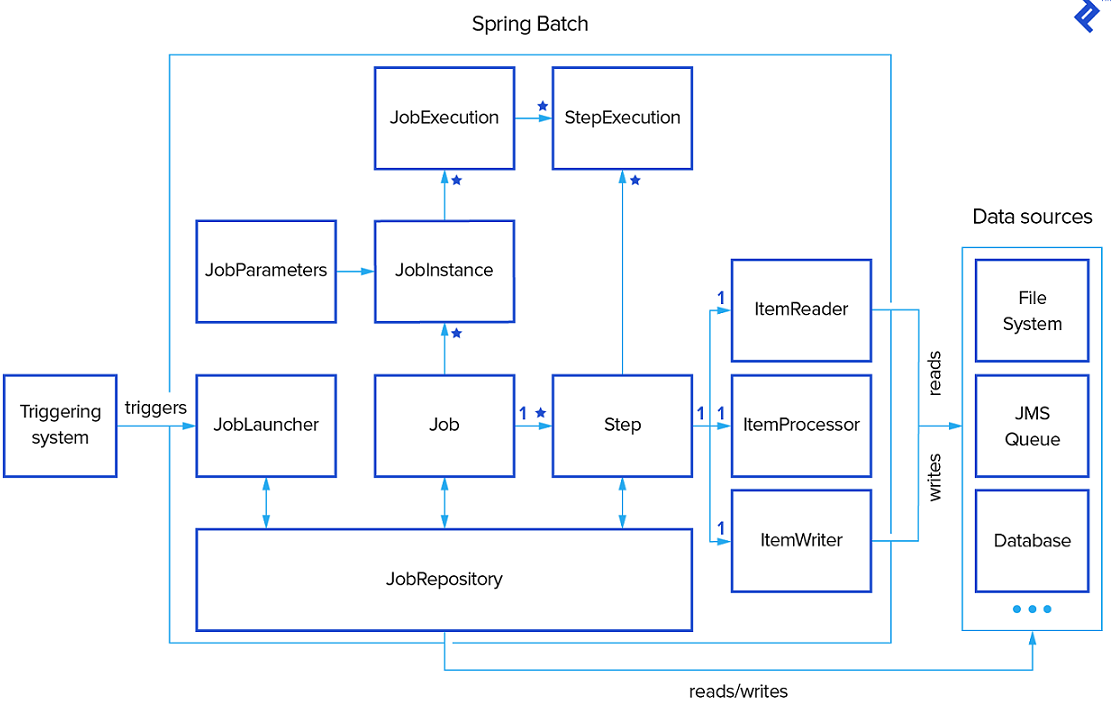
**Spring Boot: -**

1. Spring Boot is a framework that extends the **Spring framework**. Spring Boot simplifies the Spring framework by supplying **pluggable dependencies** such as Spring Batch, Spring Web Services, Spring Security, and others.
2. Spring Boot is gaining popularity because it uses **Java** as its programming language and allows developers to quickly build enterprise-grade applications with minimal configuration.
3. Spring Boot applications are auto configurable, which means that they can be configured via a list of dependencies or simply by a properties file. For example, if you list MySQL as a dependency, the Spring Boot application will start with MySQL connector included, which allows you to work seamlessly.
4. Spring Boot Framework reduces the overall development time and increases the team’s productivity by avoiding boilerplate code, configurations, and many more.
5. Spring Boot application can run on standalone mode, i.e., on one’s machine/laptop. Users don’t need a separate webserver to deploy the application.

**Spring Boot Batch: -**

Spring boot batch processing is the automated processing of large amounts of data without the need for human intervention thus making it simpler and easier.

1. Spring Boot Batch provides the concept of a Job. So, developer can create multiple Batch Jobs for different functions and register them in the Job Repository with simple annotations.
2. Each job can be invoked or scheduled separately within a project.
3. Each Job has a series/sequence of steps to achieve the batch processing business logic.
4. Each step will include tasks such as reading, processing, and writing.
5. It provides re-usability of functions or steps which could be annotated to be re-used for different Batch Job if needed.
6. For the reader, writer, and processor tasks it provides ready made classes/templates which could be extended or used as is for operations related to database, files etc.
7. Spring Boot Batch includes reusable functions such as logging/tracing, transaction management, job processing statistics, job restart, skip, and resource management that are necessary when processing large volumes of records.
8. It also provides more advanced technical services and features that like **partitioning** **techniques**, **parallel Flows** etc which will enable extremely high-volume and high-performance batch jobs. So, developers do not have to worry about multi-threading and complex transaction managements.



# Basic Project Set up and Requirements

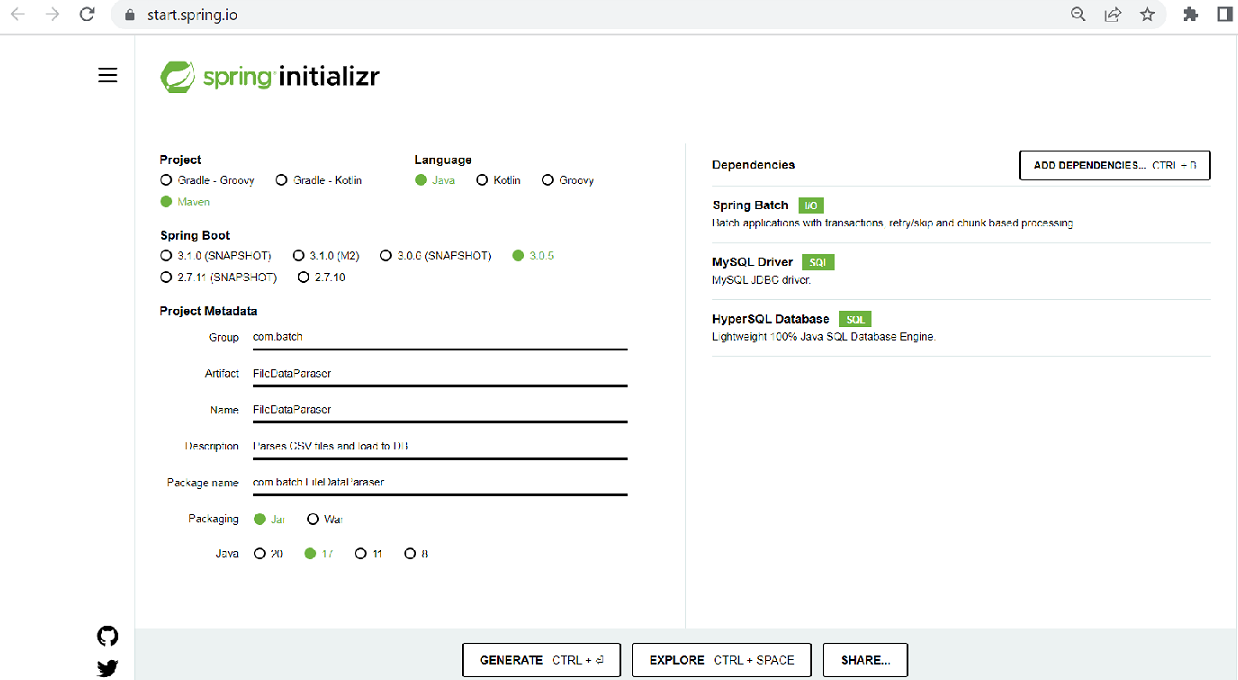
* Java 17 or higher
* Apache Maven 3.8.6 or higher
* Spring framework 6.0
* Spring Boot 3.0

We can configure the Spring Boot Batch project using Spring starter. <https://start.spring.io/>

You can configure a simple Spring Boot project using the Spring starter. This will download a basic project with required Spring Boot dependencies, on top of which developer can build the code.

Below is the major Spring Batch dependency in pom.xml.

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

****

# Basic Annotations and classes

1. **Main Class**

The default spring boot **main** class is used to begin the spring boot batch.

|  |
| --- |
| package com.batch.FileParser;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  **@SpringBootApplication**  public class SpringBootBatch2Application {  public static void main(String[] args) {  SpringApplication.run(SpringBootBatch2Application.class, args);  } |

1. **Configuration properties**

We can place our environment/profile specific properties files under below location. When invoking the Spring Batch application by passing the specific profile (eg: dev or qa or prod} the respective properties file will be applied/used.

*src/main/resources/****application-{profile}.properties***

*eg:- src/main/resources/****application-dev.properties***

We can add constant values in the properties file as below.

file.input=employee-details-report.csv

1. **Job Configuration Class**

* @**Configuration** – This annotation defines the Configuration class. This indicates that the class has bean definitions (@Bean methods). So, Spring container can process the class and generate Spring Beans to be used in the application.
* **@Bean -** This annotation is applied on a method to specify that it returns a bean to be managed by Spring context. Spring Bean annotation is usually declared in Configuration classes methods.
* @**EnableBatchProcessing** – This gives access to important beans and batch features. Annotation Interface EnableBatchProcessing enables Spring Batch features and provide a base configuration for setting up batch jobs in an @Configuration class.
* @**Autowire -** Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

|  |
| --- |
| @Configuration  @EnableBatchProcessing  public class **BatchConfiguration** {  @Autowired  public JobBuilderFactory jobBuilderFactory;  @Autowired  public StepBuilderFactory stepBuilderFactory;  @Value("${file.input}")  private String fileInput;  @Bean  public Employee employeeDetails(){    return new Employee();  }    } |

Internally Spring can identify the method by bean id :-

|  |
| --- |
| // Using AnnotationConfigApplicationContext  // instead of ClassPathXmlApplicationContext  // because we are not using XML Configuration  ApplicationContext context  = new AnnotationConfigApplicationContext(  CollegeConfig.class);  // Getting the bean  Employee empObj  = context.getBean("employeeDetails", Employee.class); |

Let us define the Job and Steps in the above configuration class: **BatchConfiguration.java**

* The autowired **JobBuilderFactory** class generates a batch task.
* The autowired **StepBuilderFactory** class generates a **batch step**. The batch steps will be executed by the batch job.
* **Batch jobs** such as **ItemReader**, **ItemProcessor**, and **ItemWriter** will be defined in the batch step. The spring boot batch **configuration** defines how the batch should be run.

|  |
| --- |
| **//Job Object**  **@Bean**  **public Job fileParser(){**  **return** jobBuilderFactory**.get("fileParser")**  **.incrementer(new RunIdIncrementer())**  **.listener(FileParserlistener())**  **.start(stepProcessFile())**  .**next(stepDeleteFile())**  // .next(stepC())  **.build()**  **;**  **}**  **//Step Object**  **@Bean**  **public Step stepProcessFile () {**  **return** stepBuilderFactory**.get("stepProcessFile ")**  **.<Employee,Employee>chunk(200)**  **.reader(reader())**  **.processor(processor())**  **.writer(writer())**  **.build()**  **;**  **}** |

1. **Other Common Annotations**

* **@ComponentScan (basePackageClasses = Employee. Class)**

Spring needs to know which packages to scan for annotated components to add them to the IoC container. In a Spring Boot project, we typically set the main application class with the @**SpringBootApplication** annotation. Under the hood, @SpringBootApplication is a composition of the @**Configuration**, @**ComponentScan**, and @**EnableAutoConfiguration** annotations. With this default setting, Spring Boot will auto scan for components in the current package (containing the @SpringBoot main class) and its sub packages.

Note: It is recommended that you locate your main application class in a root package above the component classes of the application.

The @ComponentScan annotation is used with the @Configuration annotation to tell Spring the packages to scan for annotated components. @ComponentScan also used to specify **basePackages** and **basePackageClasses** using the base Package Classes or base Packages attributes of @ComponentScan.

The **basePackageClasses** attribute is a type-safe alternative to basePackages. When you specify **basePackageClasses**, Spring will scan the **package** (and **subpackages**) of the classes you specify.

|  |
| --- |
| @Configuration  @ComponentScan(basePackages = {  "com.batch.fileparser.controller",  " com.batch.fileparser.rest”  },  basePackageClasses = FileParserController.class)  public class **BatchConfiguration** { |

* **@Controller -** The @Controller annotation indicates that a particular class serves the role of a controller.
* **@Component -** Spring Component annotation is used to denote a class as Component. It means that Spring framework will autodetect these classes for dependency injection when annotation-based configuration and class path scanning is used.
* **@Service** - Service annotation is used in your service layer and annotates classes that perform service tasks or business logic.
* **@Entity** - This annotation specifies that the class is an entity and is mapped to a database table.
* **@Repository –** This annotation is a specialization of **@Component** annotation which is used to indicate that the class provides the mechanism for **storage, retrieval, update, delete and search operation** on objects. Though it is a specialization of @Component annotation, so Spring Repository classes are autodetected by spring framework through classpath scanning. This annotation is a general-purpose stereotype annotation which very close to the [DAO pattern](https://www.geeksforgeeks.org/data-access-object-pattern/) where DAO classes are responsible for providing CRUD operations on database tables.

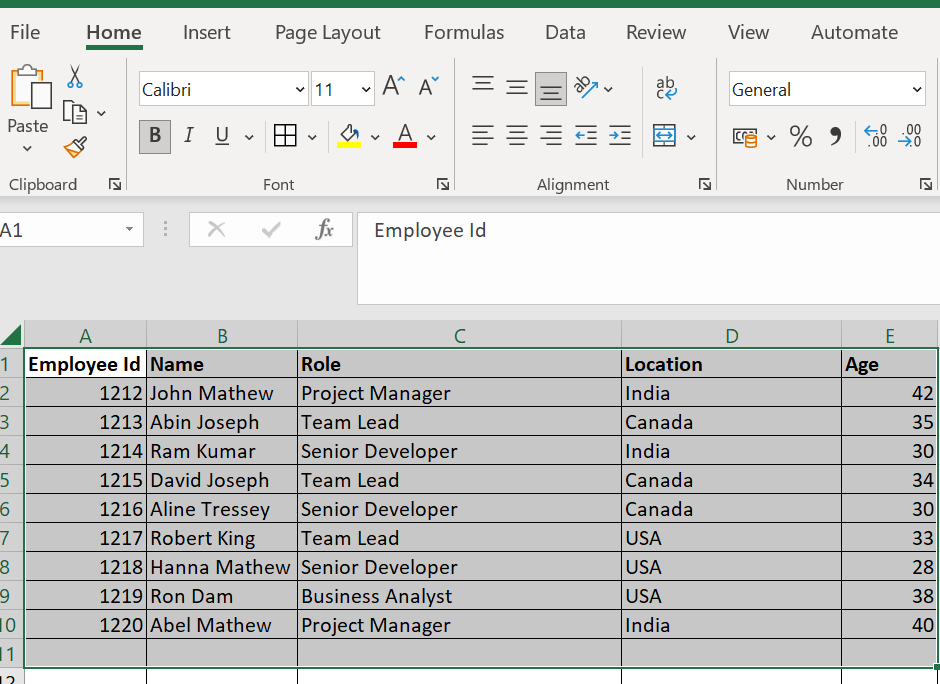
|  |
| --- |
| @Repository  public class EmployeeRepository implements BaseRepository<Employee> {    private Map<Long, Employee> repository;  public EmployeeRepository() {  this.repository = new HashMap<>();  }  // Implementation for save method  @Override  public void save(Employee employee) {  repository.put(employee.getId(),employee);  }  // Implementation for findEmployeeById method  @Override  public Student findmEmployeeById(Long id) {  return repository.get(id);  }  } |

# Spring Boot Batch Code Components

After going through above topics, we understood that in Spring Batch we can create different job(s) in a configuration class. Each job can be invoked/scheduled individually from any external tool by specifying the job name. The job encloses a sequence of **steps(methods)** or **Tasklet** to achieve the batch task.

Taking an example of small Spring Batch job which reads data from an external CSV file and processes and writes the data to the MySQL database.

Let us consider CSV file in below format in our shared local drive (D://data/)



1. **Application Properties**

We can add the file path details in the **application.properties** file. This is covered in next section in detail.

|  |
| --- |
| input.file.name=employee-details-report.csv  input.file.folder =D:/data/ |

1. **Table, Entity Class and Repository Class**

We can create an **Employee** table in **MySql** **database** where we will store this CSV data.

|  |  |  |
| --- | --- | --- |
| **EMPLOYEE** |  |  |
| **empId** | Long | Primary Key |
| **name** | Varchar | Not Null |
| **role** | Varchar | Not Null |
| **location** | Varchar | Not Null |
| **age** | Integer | Not Null |

Define an **Entity** class for the **Employee** table.

|  |
| --- |
| **import javax.persistence.Entity;**  **import javax.persistence.GeneratedValue;**  **import javax.persistence.Id;**  **import lombok.AllArgsConstructor;**  **import lombok.Data;**  **import lombok.NoArgsConstructor;**  **@Entity**  **@Data**  **@NoArgsConstructor**  **@AllArgsConstructor**  **public class Employee {**  **@Id**  **private Long employeeId;**  **private String name;**  **private String role;**  **private String location;**  **private Integer age;**  **}** |

Create a JPA repository interface for the Employee table operations.

|  |
| --- |
| **import org.springframework.data.jpa.repository.JpaRepository;**  **Import com.dev.springboot.entity.Invoice;**  @Repository  **public interface EmployeeRepository extends JpaRepository<Employee, Long>{**  **}** |

1. **Configuration class with the Job and Step method**

The configuration class- **BatchConfiguration.java** will have the major annotations **@Configuration** and **@EnableBatchProcessing**.

* **@EnableBatchProcessing** enables the class for the batch operation and wires the **JobBuilderFactory** and **StepBuilderFactory** configurations.
* Define the Job method for your batch as below.
* Define the Step(s) under the job as below.

|  |
| --- |
| **import org.springframework.batch.core.Job;**  **import org.springframework.batch.core.JobExecutionListener;**  **import org.springframework.batch.core.Step;**  **import org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;**  **import org.springframework.batch.core.configuration.annotation.JobBuilderFactory;**  **import org.springframework.batch.core.configuration.annotation.StepBuilderFactory;**  **import org.springframework.batch.core.launch.support.RunIdIncrementer;**  **@Configuration**  **@EnableBatchProcessing**  **public class BatchConfiguration {**  **//Autowire JobBuilderFactory**  **@Autowired**  **private JobBuilderFactory jobBuilderFactory;**  **//Autowire StepBuilderFactory**  **@Autowired**  **private StepBuilderFactory stepBuilderFactory;**  **//Autowire InvoiceRepository**  **@Autowired**  **EmployeeRepository repository;**  **//Read the file path from properties file**  **@Value("${**input.file.name **}")**  **private String fileName;**  **@Value("${**input.file.folder**}")**  **private String fileLocation;**  **//Job Object**  **@Bean**  **public Job fileParser(){**  **return jobBuilderFactory.get("fileParser")**  **.incrementer(new RunIdIncrementer())**  **.listener(listener())**  **.start(stepProcessFiles())**  **.next(stepDeleteFiles())**  // .next(stepC())  **.build()**  **;**  **}**  **//Step Object**  **@Bean**  **public Step stepProcessFiles() {**  **return stepBuilderFactory.get("stepProcessFiles ")**  **.<Employee, Employee >chunk(2)**  **.reader(reader())**  **.processor(processor())**  **.writer(writer())**  **.build()**  **;**  **}**  --  --  --  } |

1. **Reader, Processor and Writer**

**Reader**

* + An Item Reader reads data into the spring batch application from a particular source.
  + It is the entity of a step (of a batch process) which reads data. An **ItemReader** reads one item a time.
  + Spring Batch provides an Interface **ItemReader**. All the readers **implement** this **interface**.

|  |  |
| --- | --- |
| **Reader** | **Purpose** |
| FlatFIleItemReader | To read data from flat files. |
| StaxEventItemReader | To read data from XML files. |
| StoredProcedureItemReader | To read data from the stored procedures of a database. |
| JDBCPagingItemReader | To read data from relational databases database. |
| MongoItemReader | To read data from MongoDB. |
| Neo4jItemReader | To read data from Neo4jItemReader. |

Define the reader () method bean

|  |
| --- |
| **//Reader class Object**  **@Bean**  **public FlatFileItemReader<Employee> reader() {**  **String filePath= fileLocation+ filename;**  **FlatFileItemReader< Employee > reader= new FlatFileItemReader<>();**  **reader.** **setResource**  **(new FileSystemResource(filePath));**  **reader.setLineMapper(new DefaultLineMapper<>() {{**  **setLineTokenizer(new DelimitedLineTokenizer() {{**  **setDelimiter(*DELIMITER\_COMMA*);**  **setNames(“employeeId” "name","role","location","age");**  **}});**  **setFieldSetMapper(new BeanWrapperFieldSetMapper<>() {{**  **setTargetType(Employee.class);**  **}});**  **}});**  **return reader;**  **}** |

**Chunking & Page Size**

* + The chunk processing method allows for the processing of a single block of **data at a time. If the chunk is specified with a value greater than one, for** example, 2, the batch will run the read method 2 times in the **ItemReader** and process the data 2 times in the **ItemProcessor**.
  + The **ItemWriter’s** write method will only be called once, and it will include all 2 process data. All 2 processed data sets can be committed to a database or transmitted over the network in a single session.
  + We can also define optimal page size (no. of records per read) or max read size based on data volume and commit interval factors.

reader.setPageSize(10000);

reader.setMaxItemCount(50\*reader.getPageSize());

**Processor**

* + An **ItemProcessor** is used to process the data. When the given item is not valid it returns null, else it processes the given item and returns the processed result. The interface **ItemProcessor**<I,O> represents the processor.
  + We can define a custom item processor by implementing the interface **ItemProcessor** of the package **org.springframework.batch.item.ItemProcessor**. This **ItemProcessor** class accepts an object and processes the data and returns the processed data as another object.
  + In a batch process, if **"n"** records or data elements are read, then for each record, it will read the data, process it, and writes the data in the writer. To process the data, it relays on the processor passed.

|  |
| --- |
| **//Processor class Object**  **@Bean**  **public ItemProcessor<Employee, Employee > processor(){**  **return new EmployeeProcessor();**  **}** |

|  |
| --- |
| **@Component**  **public class EmployeeProcessor implements ItemProcessor<Employee, Employee> {**  **private static final Logger LOGGER = LoggerFactory.getLogger(EmployeeProcessore.class);**  **@Override**  **public Employee process(final Employee employee) throws Exception {**  **String name = employee.getName();**  **if (StringUtils.isBlank(employee.getName())) {**  **return null;**  **}**  **return employee;**  **}**  **}** |

**Writer**

* + It is the element of the step of a batch process which writes data.
  + An **ItemWriter** writes one item a time.
  + Spring Batch provides an Interface ItemWriter. All the writers implement this interface.

|  |  |
| --- | --- |
| **Writer** | **Purpose** |
| FlatFIleItemWriter | To write data into flat files. |
| StaxEventItemWriter | To write data into XML files. |
| StoredProcedureItemWriter | To write data into the stored procedures of a database. |
| JDBCPagingItemWriter | To write data into relational databases database. |
| MongoItemWriter | To write data into MongoDB. |
| Neo4jItemWriter | To write data into Neo4j. |

|  |
| --- |
| **//Writer class Object**  **@Bean**  **public ItemWriter< Employee > writer(){**  **return employeeList -> {**  **System.*out*.println("Saving Invoice Records: " + employee);**  **repository.saveAll(employeeList);**  **};**  **}** |

1. **TaskLet Class**
   * In Spring batch, the **Tasklet** is an interface, which will be called to perform a single task only, like clean or set up resources before or after any step execution.

Calling the **step** that invokes **Tasklet** in the configuration class: **BatchConfiguration.java**

|  |
| --- |
| @Bean  public Step stepDeleteFiles() {  return stepBuilderFactory.get("stepDeleteFiles")  .tasklet(new FileDeletingTasklet()).build();  } |

**Tasklet Class : FileDeletingTasklet**

|  |
| --- |
| import java.io.File;  import java.io.IOException;  import java.nio.file.Files;  import java.nio.file.Path;  import java.nio.file.Paths;  import java.util.stream.Stream;  import org.springframework.batch.core.StepContribution;  import org.springframework.batch.core.UnexpectedJobExecutionException;  import org.springframework.batch.core.scope.context.ChunkContext;  import org.springframework.batch.core.step.tasklet.Tasklet;  import org.springframework.batch.repeat.RepeatStatus;  import org.springframework.core.io.Resource;  public class FileDeletingTasklet implements Tasklet {  private Resource directory;  public FileDeletingTasklet(Resource directory) {  this.directory = directory;  }  @Override  public RepeatStatus execute(StepContribution stepContribution,  ChunkContext chunkContext) {  try (Stream<Path> walk =  Files.walk(Paths.get(directory.getFile().getPath()))) {  walk.filter(Files::isRegularFile).map(Path::toFile)  .forEach(File::delete);  } catch (IOException e) {  LOGGER.error("error deleting files", e);  throw new UnexpectedJobExecutionException(  "unable to delete files");  }  return RepeatStatus.FINISHED;  }  } |

1. **Listener Class**
   * A listener class implements Spring framework’s **JobExecutionListener**.
   * It has two methods **beforeJob()** and **afterJob()** overwritten from the interface **JobExecutionListener** as shown below.
   * This is an optional class to create if we want to execute some logic before and after the job execution.

In your configuration class (BatchConfiguration.java), define the Listener method bean

|  |
| --- |
| **//Listener class Object**  **@Bean**  **public JobExecutionListener listener() {**  **return new EmployeeListener ();**  **}** |

Create a Listener class: EmployeeListener.java

|  |
| --- |
| **import org.springframework.batch.core.JobExecution;**  **import org.springframework.batch.core.JobExecutionListener;**  **public class EmployeeListener implements JobExecutionListener{**  **@Override**  **public void beforeJob(JobExecution jobExecution) {**  **System.*out*.println("Job started at: "+ jobExecution.getStartTime());**  **System.*out*.println("Status of the Job: "+jobExecution.getStatus());**  **}**  **@Override**  **public void afterJob(JobExecution jobExecution) {**  **System.*out*.println("Job Ended at: "+ jobExecution.getEndTime());**  **System.*out*.println("Status of the Job: "+jobExecution.getStatus());**  **}**  **}** |

# Application Properties profiles, Database set up

**application-properties file**

For each environment profile we can define respective application properties files. For eg:- if we have **Dev1** environment, then we can define **application-dev1.properties**.Similarly for QA1 profile we can define **application-qa1.properties**

Full path: - src/main/resources/application-dev.properties

We have to define the database details in the application.properties file

|  |
| --- |
| #Database Connection  **spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver**  **spring.datasource.url=jdbc:mysql://localhost:3306/batchdb**  **spring.datasource.username=root**  **spring.datasource.password=\*\***  #----------ORM Details-------------------  #To display SQL At console  **spring.jpa.show-sql=true**  #To Create tables  **spring.jpa.hibernate.ddl-auto=create**  #To Generate SQL queries  **spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect**  **#----------Spring Batch Properties----------**  # By default it's true which means all the Spring batches will start executing automatically  **spring.batch.job.enabled=true**  # Tables for metadata created by Spring Boot (Always, Embedded, Never)  **spring.batch.jdbc.initialize-schema=ALWAYS**  #logging  **log4j.rootCategory=INFO, stdout**  **log4j.appender.stdout=org.apache.log4j.ConsoleAppender**  **log4j.appender.stdout.layout=org.apache.log4j.PatternLayout**  **log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %t %c{2}:%L - %m%n**  **log4j.category.org.springframework.beans.factory=DEBUG** |

**Data Source**

In the configuration class: **BatchConfiguration**.java we can define the **Data Source** and pass it to the Reader and Writer class/method.

**BatchConfiguration.java**

|  |
| --- |
| @Autowired  private DataSource dataSource;  @Autowired  Environment env;  private DataSource batchDataSource() {  return DataSourceBuilder.create()  .url(env.getProperty("spring.datasource.url "))  .driverClassName(env.getProperty("spring.datasource.driver"))  .username(env.getProperty("spring.datasource.username"))  .password(env.getProperty("spring.datasource.password"))  .build();  }  @Bean  public ItemReader<User> notifySubscribersReader(){  JdbcCursorItemReader<User> reader = new JdbcCursorItemReader<User>();  String sql = "select \* from users where is\_email\_subscriber is not null";  reader.setSql(sql);  reader.setDataSource(dataSource);  reader.setRowMapper(rowMapper());  return reader;  } |

# Spring Boot Batch – Deployment, Scheduling

Spring Boot Batch Job can be invoked/scheduled in number of ways. Few common methods are:

* Rest Controller
* Batch Script (Windows - .bat, Unix - .sh)
* External 3rd party scheduling tools.
* Cron jobs

**Rest Controller Method**

* Define a **Rest Controller** class with @RestController Annotation.
* The controller can invoke the created Job by Job name and pass the required job parameters.
* The Spring Boot Batch **JobLauncher** class will run the Job.
* Once the application is deployed we can hit the service url to trigger the batch job:- <http://localhost:8080/invokjob>

|  |
| --- |
| package com.batch.fileParser.controller;  import org.springframework.batch.core.Job;  import org.springframework.batch.core.JobParameters;  import org.springframework.batch.core.JobParametersBuilder;  import org.springframework.batch.core.launch.JobLauncher;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.web.bind.annotation.GetMapping;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RestController;  @RestController  public class SpringBatchJobController {  @Autowired  JobLauncher jobLauncher;  @Autowired  Job fileParser;  @GetMapping("/invokejob")  public String invokeBatchJob() throws Exception {  JobParameters jobParameters = new JobParametersBuilder()  .addLong("time", System.currentTimeMillis())  .toJobParameters();  jobLauncher.run(fileParser, jobParameters);  return "Batch job has been invoked";  }  } |
|  |

**Batch Script (Windows - .bat, Unix - .sh)**

* In case of standalone batch applications on a **Windows jBatch machine** or **Linux** machine, we can write simple batch script to invoke the Spring Boot batch app.

Sample batchJob.bat file for Windows

|  |
| --- |
| title data-middleware  @set JAVA\_HOME=D:\java\jdk1.8.0\_181  @set PATH=%JAVA\_HOME%\bin;%PATH%  D:  cd data-middleware\  java -jar fileParser.jar --spring.profiles.active=dev  exit |

**External 3rd party scheduling tools and Cron jobs**

* Any external 3rd party scheduler tool (For eg: Autosys) and cron job can be scheduled to invoke this batch job either by executing the script file (.bat, .sh) or by calling the Rest service.

# Spring Boot Batch – Spring framework 6 upgrade

The above batch code is for Spring Boot 2.x versions which is running on Spring Framework below 5.x version. Spring Boot v3.0 runs on the latest spring framework 6 version which requires lot of changes.

In the latest Spring framework 6 and Spring Boot version 3 there are few changes in the Batch due to certain deprecations. Refer the Spring Migration Guide for more details:-

<https://github.com/spring-projects/spring-batch/wiki/Spring-Batch-5.0-Migration-Guide>

Below are few high-level changes: -

* 1. Minimum **Java version 17**. Upgrade the Java version and Maven compiler version (if specified) in pom. Xml.

|  |
| --- |
| <properties>  <java.version>17</java.version>  <**maven.compiler.target**>17</**maven.compiler.target**>  <**maven.compiler.source**>17</**maven.compiler.source**>  </properties> |

* 1. **Deprecation** of **JobBuilderFactory** and **StepBuilderFactory**.
  + Spring Boot 3.x has deprecated the JobBuilderFactory and StepBuilderFactory.
  + The JobBuilder and StepBuilder class is used for reading the Job and Steps.
  + The JobRepository and Step name can be passed to the Job method as parameter.

|  |
| --- |
| **import** org.springframework.batch.core.job.builder.JobBuilder;  @Bean  **public** Job job(JobRepository jobRepository,JobExecutionListener listener,Step stepProcessFile) {  System.***out***.println("Job started: ");  **return** **new** JobBuilder("fileparserjob",jobRepository)  .incrementer(**new** RunIdIncrementer())  .listener(listener)  .flow(stepProcessFile(**null**,**null**))  .end()  .build();  }  @Bean  **public** Step stepProcessFile(JobRepository jobRepository,PlatformTransactionManager transactionManager) {  **return** **new** StepBuilder("stepProcessFile",jobRepository)  .<Product, Product>chunk(2,transactionManager)  .reader(productReader())  .processor(processor())  .writer(productCustomWriter)  .build();  } |

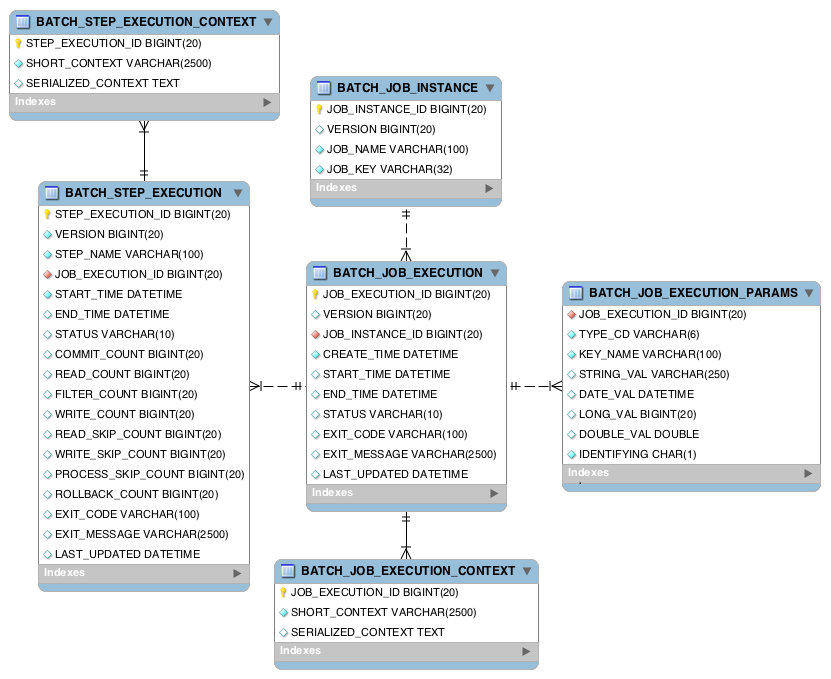
* 1. **Deprecation** of the **JobExecutionListenerSupport** class.
  + The **JobExecutionListenerSupport** class is deprecated.
  + The custom Listener class which we writer can directly implement the **JobExecutionListener** interface.

|  |
| --- |
| **import** org.springframework.batch.core.JobExecution;  **import** org.springframework.batch.core.JobExecutionListener;  **import** org.springframework.stereotype.Component;  @Component  **public** **class** FileParserJobListener **implements** JobExecutionListener {    @Override  **public** **void** beforeJob(JobExecution jobExecution) {  System.***out***.println("Job started at: "+ jobExecution.getStartTime());  System.***out***.println("Status of the Job: "+jobExecution.getStatus());  }  @Override  **public** **void** afterJob(JobExecution jobExecution) {  System.***out***.println("Job Ended at: "+ jobExecution.getEndTime());  System.***out***.println("Status of the Job: "+jobExecution.getStatus());  }  } |

* 1. On upgrading the Spring Boot parent version, the dependencies will get auto-updated unless we have specified enforced <version>/<version>.
  + Any version which the developer has enforced needs to be checked and upgraded version needs to be provided which is compatible with Spring Framework 6.
  + Java EE has been moved to Jakarta EE 9. Not owned by Oracle but handed to Eclipse Foundation-open source.) :-
  + Code changes in all entity classes, JPA repositories - to change imports from javax.\* to Jakarta.\* and related code changes.
  + Upgrade the Hibernate ORM to 6.x (JPA 3.x based). All related Hibernate jars.
  + Each service code files will have to checked for javax references and changed with Jakarta. Example. JSON libraries
  + Changing the Java EE (8) to Jakarta EE (9). Eg: JMS to Jakarta Messaging, JSP to Jakarta Server Pages.

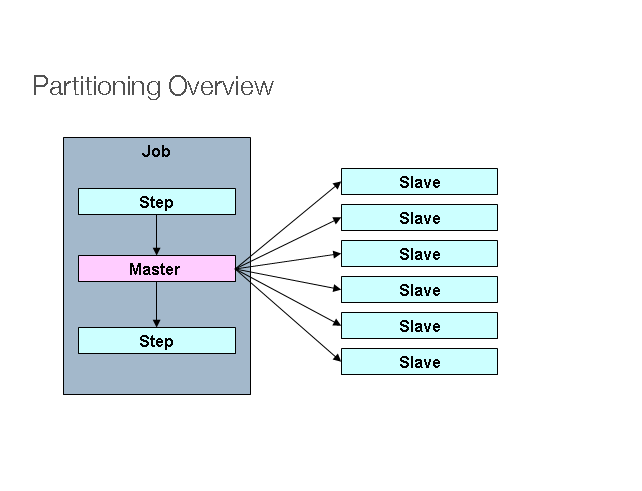
# Spring Boot Batch – OOB Tables

* The Spring Batch **Metadata tables** closely match the **domain** objects that represent them in Java. For example, **JobInstance**, **JobExecution**, **JobParameters**, and **StepExecution** map to **BATCH\_JOB\_INSTANCE, BATCH\_JOB\_EXECUTION, BATCH\_JOB\_EXECUTION\_PARAMS, and BATCH\_STEP\_EXECUTION**, respectively.
* Support team can monitor these tables to know the audit details of any job.



# Parallel Processing- Partition and Flow

1. **Partitioning**



* Manager is a Step that has been partitioned into multiple Worker steps that are also instances of Step. Workers can be some remote services, locally executing threads or any other independent tasks.
* Spring batch allows passing the input data from Manager to Worker steps so each worker knows exactly what to do. The JobRepository ensures that every worker is executed only once in a single execution of the Job
* Partitioning is useful when we have millions of records to read from source systems, and we can’t just rely on a single thread to process all records, which can be time-consuming. We want to use multiple threads to read and process data to use system resources effectively.

|  |
| --- |
| @Bean  public Step partitionStep()  throws UnexpectedInputException, MalformedURLException, ParseException {  return steps.get("partitionStep")  .partitioner("stepProcessFiles ", partitioner())  .step(stepProcessFiles())  .taskExecutor(taskExecutor())  .build();  }  @Override  public Map<String, ExecutionContext> partition(int gridSize) {  Map<String, ExecutionContext> map = new HashMap<>(gridSize);  int i = 0, k = 1;  for (Resource resource: resources) {  ExecutionContext context = new ExecutionContext();    context.putString(fileName, resource.getFilename()+ ".csv");    map.put(PARTITION\_KEY + i, context);  i++;  }  return map;  }  } |

|  |
| --- |
| **@Bean**  **public Step stepProcessFiles() {**  **return stepBuilderFactory.get("stepProcessFiles ")**  **.<Employee, Employee >chunk(2)**  **.reader(reader(null))**  **.processor(processor())**  **.writer(writer())**  **.build()**  **;**  **}** |

|  |
| --- |
| **//Reader class Object**  **@Bean**  **public FlatFileItemReader<Employee> reader(**@Value("#{stepExecutionContext[fileName]}") String filename**) {**    **FlatFileItemReader< Employee > reader= new FlatFileItemReader<>();**  **reader.** **setResource**  **(new FileSystemResource(fileName));**  **reader.setLineMapper(new DefaultLineMapper<>() {{**  **setLineTokenizer(new DelimitedLineTokenizer() {{**  **setDelimiter(*DELIMITER\_COMMA*);**  **setNames(“employeeId” "name","role","location","age");**  **}});**  **setFieldSetMapper(new BeanWrapperFieldSetMapper<>() {{**  **setTargetType(Employee.class);**  **}});**  **}});**  **return reader;**  **}** |

1. **Flows**

Flows can be of 2 types: -

* 1. Parallel Flows:- The steps defined in 2 parallel flows will run in parallel.
  2. Conditional Flows:- The flow which meets the condition will be triggered and the other flow will not be invoked.

# References

<https://javatechonline.com/spring-boot-batch-example-csv-to-mysql-using-jpa/>

<https://youtu.be/_TSjkSn2yvQ>

<https://github.com/Java-Techie-jt/spring-batch-3.0>

<https://www.dineshonjava.com/spring-batch-read-from-csv-and-write-to-relational-db/>

<https://www.jackrutorial.com/2018/03/spring-boot-batch-read-from-mysql-database-and-write-into-a-csv-file-tutorial.html>

<https://www.tutorialspoint.com/spring_batch/spring_batch_readers_writers_processors.htm>

<https://www.javadevjournal.com/spring-batch/spring-batch-java-configuration/>

<https://docs.spring.io/spring-batch/docs/current/reference/html/schema-appendix.html>

<https://www.bezkoder.com/spring-boot-upload-csv-file/>

<https://github.com/spring-projects/spring-batch/wiki/Spring-Batch-5.0-Migration-Guide>