Spring Batch

Development:

Marcelo Eduardo Guillen Castillo

Java Monterrey Academy

September 5,, 2024

Introduction **3**
Business case 3
Objectives 3
Requirements **3**
Installation **3**
Project architecture **5**
Demonstrations **6**
Conclusion **9**

Introduction

Business case

Our applications sometimes need modifications, updates and reports in order to verify if everything is correctly done, and on Spring Boot this task couldn't be more easier. Spring Batch is a dependency for the Spring Boot framework which can create batch processing, the capacity to process by tasks or jobs, processing of a lot of data and information, normally could be used as middleware for the principal app. Our company is recently needed to move books information from Excel into the recently database created by the team B (the Spring Data JPA team), and the company need our help, team C, to implement for this service a batch processing service for reporting these new books from the CSV files into the database data, also making a log register of its movements.

Objectives

In order to move all our manually typed data from the Excel files into the database without the need of any human or manual interference.
Finally move completely into the service technology, leaving behind the Excel long sheets and trivial problems with data.

Requirements

- Read all data from the CSV file
- Could make the processing from a get method
- Write all data into the database
- Process the data, by filtering year and genre.

Installation

The engineer needs to install the JDK 17 to beyond, then any IDE that could maintain Maven projects, and finally install all the dependencies and plugins

necessary to make the project work correctly. This is made in Maven, so please enter the following code into your POM file.

```
Unset
<project xmlns="http://maven.apache.org/POM/4.0.0"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>2.6.7</version>
		<relativePath /> <!-- lookup parent from repository -->
	</parent>
	<groupId>group</groupId>
	<artifactId>MyOwnSpringBatch</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-batch</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-data-jpa</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>

		<dependency>
			<groupId>mysql</groupId>
			<artifactId>mysql-connector-java</artifactId>
			<scope>runtime</scope>
		</dependency>
		<dependency>
			<groupId>org.projectlombok</groupId>
			<artifactId>lombok</artifactId>
			<optional>true</optional>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
		</dependency>
		<dependency>
```

```
                    <groupId>org.springframework.batch</groupId>
                    <artifactId>spring-batch-test</artifactId>
                    <scope>test</scope>
             </dependency>
      </dependencies>

      <build>
             <plugins>
                    <plugin>
                           <groupId>org.springframework.boot</groupId>
                           <artifactId>spring-boot-maven-plugin</artifactId>
                           <configuration>
                                  <excludes>
                                         <exclude>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>
                                         </exclude>
                                  </excludes>
                           </configuration>
                    </plugin>
             </plugins>
      </build>
</project>
```

Also we use a database on MySQL, being modified with MySQL Workbench, but you can use any kind of IDE for MySQL databases.
We use postman for our manual tests, a software for web service testing, but you can use any tool or extension you want in order to verify the correct functionalities of the service.

Project architecture

Our library can handle books as its data, and has the following characteristics:

- ID: identification number.
- Title: name of the book.
- Author: name of the author.
- Year: publication years.
- Genre: type of genre of the book.

- Is read: specifies if the book was being read.

Book
+ id: int(11)
+ title: varchar(64)
+ author: varchar(64)
+ year: int(11)
+ genre: varchar(64)
+ is_read: boolean

Demonstrations

Here are some code demonstrations, some of the implementations about functionalities of the software.
First of all we create a Main class with its typical main method, then we declare to the compiler that our Main class will be the Spring Boot Application by annotation, and then run the Spring Application with the Main class and its arguments.

```
 6 @SpringBootApplication¤¶
 7 public·class·Main·{¤¶
 8 ¤¶
 9●»   public·static·void·main(String[]·args)·{¤¶
10 »   »   SpringApplication.run(Main.class,·args);¤¶
11 »   }¤¶
12 ¤¶
13 }¤¶
14
```

Then we create a class Book to be intended as a Spring Entity, with its attributes.

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name = "BOOK_INFO")
public class Book {
    @Id
    @Column(name = "ID")
    private int id;
    @Column(name = "TITLE")
    private String title;
    @Column(name = "AUTHOR")
    private String author;
    @Column(name = "YEAR")
    private int year;
    @Column(name = "GENRE")
    private String genre;
    @Column(name = "IS_READ")
    private boolean read;
}
```

Then we create the Book Repository.

```java
public interface BookRepository extends JpaRepository<Book, Integer> {
}
```

We need to create a configuration class for our jobs and steps, it will need a job builder factory, a step builder factory and a book repository.

```java
@Configuration
@EnableBatchProcessing
@AllArgsConstructor
public class Config {
    private JobBuilderFactory jobBuilderFactory;

    private StepBuilderFactory stepBuilderFactory;

    private BookRepository repo;

```

We need some functions that will be doing important tasks during the process execution.

A line mapper that will be in charge to tell the service how the file should be read and which information will be used as reference to insert the read data into the process.

```
36 //» We specified how file should be read and mapped¤¶
37●» private LineMapper<Book> lineMapper() {¤¶
38 » » DefaultLineMapper<Book> mapper = new DefaultLineMapper<>();¤¶
39 ¤¶
40 » » DelimitedLineTokenizer dlt = new DelimitedLineTokenizer();¤¶
41 » » dlt.setDelimiter(",");¤¶
42 » » dlt.setStrict(true);¤¶
43 » » dlt.setNames("ID", "TITLE", "AUTHOR", "YEAR", "GENRE", "READ");¤¶
44 ¤¶
45 » » BeanWrapperFieldSetMapper<Book> setMapper = new BeanWrapperFieldSetMapper<Book>();¤¶
46 » » setMapper.setTargetType(Book.class);¤¶
47 ¤¶
48 » » mapper.setLineTokenizer(dlt);¤¶
49 » » mapper.setFieldSetMapper(setMapper);¤¶
50 » » return mapper;¤¶
51 » }¤¶
```

Then we need some important items in order to make the step work correctly.
A reader item for input data, in this case, for reading a csv file.

```
62 //» The ItemReader of the step for the file lecture¤¶
63●» @Bean¤¶
64 » public FlatFileItemReader<Book> reader() {¤¶
65 » » FlatFileItemReader<Book> itemReader = new FlatFileItemReader<>();¤¶
66 » » itemReader.setResource(new FileSystemResource("src/main/resources/books.csv"));¤¶
67 » » itemReader.setName("csvReader");¤¶
68 » » itemReader.setLinesToSkip(1);¤¶
69 » » itemReader.setLineMapper(lineMapper());¤¶
70 » » return itemReader;¤¶
71 » }¤¶
```

A writer item for output data, in this case, for writing the processed data into the database, telling which repository to use and which method to implement.

```
53 //» The ItemWriter of the process we are going to write the information into the database ¤¶
54●» @Bean¤¶
55 » public RepositoryItemWriter<Book> writer() {¤¶
56 » » RepositoryItemWriter<Book> w = new RepositoryItemWriter<>();¤¶
57 » » w.setRepository(repo);¤¶
58 » » w.setMethodName("save");¤¶
59 » » return w;¤¶
60 » }¤¶
```

We need a task executor in order to tell the step to execute all the steps in an asynchronous way, telling him in this case that we want a maximum of 10 concurrent procesess.

```
84 //» We configure a maximum of 10 concurrent processes for our data¤¶
85●» @Bean¤¶
86 » public TaskExecutor taskExecutor() {¤¶
87 » » SimpleAsyncTaskExecutor sate = new SimpleAsyncTaskExecutor();¤¶
88 » » sate.setConcurrencyLimit(10);¤¶
89 » » return sate;¤¶
90 » }¤¶
```

Then we specified the step for our job, here we are going to make an ItemProcessor using functional programming, instead of creating a separate interface for that ItemProcessor, the filter just consists on execute the writing only if the genre is equals to “Science Fiction” and the year of the book is above or equal to 2023.

```
// The step that is going to read the file, process all data and write new data to the database
    @Bean
    public Step paso1() {
        return stepBuilderFactory.get("csv-step").<Book, Book>chunk(10).reader(reader())
                .processor((ItemProcessor<Book, Book>) book -> {
                    if (book.getGenre().equals("Science Fiction") && book.getYear() >= 2023)
                        return book;
                    return null;
                }).writer(writer()).taskExecutor(taskExecutor()).build();
    }

```

We need to run the job, we can name the job.

```
// Execute all the steps for our Job
    @Bean
    public Job runJob() {
        return jobBuilderFactory.get("importBooks").flow(paso1()).end().build();
    }
}

```

Finally, we create our Rest Controller with the singular http method of importing the books from the CSV file into the database.
In this case we need to inject only a Job and a JobLauncher.

```
@RestController
@RequestMapping("/books")
public class JobController {
    @Autowired
    private JobLauncher jobLauncher;

    @Autowired
    private Job job;

    @PostMapping("/importBooks")
    public void importCsvToDBJob() {
        JobParameters params = new JobParametersBuilder()
                .addLong("startAt", System.currentTimeMillis()).toJobParameters();
        try {
            jobLauncher.run(job, params);
        } catch(JobExecutionAlreadyRunningException | JobRestartException | JobInstanceAlreadyCompleteException |
JobParametersInvalidException e) {
            e.printStackTrace();
        }

    }
}

```

Conclusion

Managing a lot of data it's just a big problem, even worst if the company needs some improvements and periodic reports of data, so it is very important tools like Spring Batch that can manage by itself tasks that need to be periodically updated and

require to be implemented on the best moments of the service, in order to avoid interrupting the traffic on our principal service.