

# 1. TP - spring-batch

## 1.1. Configuration d'un projet spring-batch

Créer un nouveau projet springBatch via spring initializr (<https://start.spring.io/>).

**Java >=17 ou 21, maven , SpringBoot 3.x (stable) .**

Group : **tp** , Atifact : **tpSpringBatch** , packaging jar

Dependencies : SpringBatch et h2 (et mySql)

Extraire dans un répertoire de travail (ex : **c:\tp**) le contenu du .zip généré et téléchargé.

Charger le projet dans votre IDE favori (IntelliJ , eclipse , VSCode ou autre) .

Ajouter la configuration Suivante dans **application.properties** :

```
spring.batch.jdbc.initialize-schema=always
spring.datasource.url=jdbc:h2:~/jobRepositoryDb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
```

Créer le nouveau package "**tp.tpSpringBatch.config**"

Au sein de ce package, créer la classe **AutomaticSpringBootBatchJobRepositoryConfig.java** en s'inspirant du modèle du support de cours.

## 1.2. Programmation et démarrage d'un job très simple

Créer le nouveau package "**tp.tpSpringBatch.tasklet**"

Au sein de ce package , coder en premier la classe suivante **PrintMessageTasklet** :

```
package tp.tpSpringBatch.tasklet;

import org.springframework.batch.core.StepContribution;
import org.springframework.batch.core.scope.context.ChunkContext;
import org.springframework.batch.core.step.tasklet.Tasklet;
import org.springframework.batch.repeat.RepeatStatus;

//no annotation , to use from xml config or in a annotated subclass in .bean subpackage
public class PrintMessageTasklet implements Tasklet{
    private String message;

    @Override
    public RepeatStatus execute(StepContribution contribution, ChunkContext chunkContext)
            throws Exception {
        System.out.println(message);
        return RepeatStatus.FINISHED;
    }

    public PrintMessageTasklet(String message) {
        this.message = message;
```

```

    }

    public PrintMessageTasklet() {
        super();
    }

    public String getMessage() { return message;}
    public void setMessage(String message) {this.message = message;}
}

```

Créer le nouveau package "**tp.tpSpringBatch.tasklet.bean**"

Au sein de ce package, coder en premier la classe suivante **PrintHelloWorldMessageTaskletBean**:

```

package tp.tpSpringBatch.tasklet.bean;
import org.springframework.stereotype.Component;
import tp.tpSpringBatch.tasklet.PrintMessageTasklet;

```

**@Component**

```

public class PrintHelloWorldMessageTaskletBean extends PrintMessageTasklet{
    public PrintHelloWorldMessageTaskletBean(){
        super("hello world by SpringBatch");
    }
}

```

Créer le nouveau package "**tp.tpSpringBatch.job.java**"

Au sein de ce package , coder en premier la classe suivante **MyAbstractJobConfig** :

```

package tp.tpSpringBatch.job.java;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.job.builder.JobBuilder;
import org.springframework.batch.core.repository.JobRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.transaction.PlatformTransactionManager;

public abstract class MyAbstractJobConfig {

    @Autowired
    protected JobRepository jobRepository;

    @Autowired
    protected PlatformTransactionManager batchTxManager;

    //NB: jobRepository will be useful in Job concrete SubClass to build new Job and new Steps
    // batchTxManager will be useful in Job concrete SubClass to build new Steps

    protected Job buildMySingleStepJob(String jobName, Step singleStep) {
        var jobBuilder = new JobBuilder(jobName, jobRepository);
        return jobBuilder.start(singleStep)
            .build();
    }
}

```

```

    }
}
```

Coder ensuite la sous classe suivante **HelloWorldJobConfig** :

```

package tp.tpSpringBatch.job.java;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.step.builder.StepBuilder;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

import tp.tpSpringBatch.tasklet.bean.PrintHelloWorldMessageTaskletBean;

@Configuration
@Profile("!xmlJobConfig")
public class HelloWorldJobConfig extends MyAbstractJobConfig {

    public static final Logger logger = LoggerFactory.getLogger(HelloWorldJobConfig.class);

    @Bean(name="myHelloWorldJob")
    public Job myHelloWorldJob(Step printMessageStepWithTasklet) {
        var name = "myHelloWorldJob";
        return this.buildMySingleStepJob(name, printMessageStepWithTasklet);
    }

    @Bean
    public Step simplePrintMessageStep(PrintHelloWorldMessageTaskletBean printHelloWorldMessageTaskletBean) {
        var name = "simplePrintMessageStep";
        var stepBuilder = new StepBuilder(name, jobRepository);
        return stepBuilder
            .tasklet(printHelloWorldMessageTaskletBean, this.batchTxManager)
            .build();
    }
}
```

Compléter enfin le code de la classe principale **TpSpringBatchApplication** avec :

- un constructeur pour injecter jobLauncher et applicationContext
- une implémentation de l'interface CommandLineRunner
- l'exécution du job "myHelloWorldJob"

```
package tp.tpSpringBatch;
```

```

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.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class TpSpringBatchApplication implements CommandLineRunner{

    private final JobLauncher jobLauncher;
    private final ApplicationContext applicationContext;

    @Autowired
    public TpSpringBatchApplication(JobLauncher jobLauncher,
                                    ApplicationContext applicationContext) {
        //injection by constructor
        this.jobLauncher = jobLauncher;
        this.applicationContext = applicationContext;
    }

    public static void main(String[] args) {
        SpringApplication.run(TpSpringBatchApplication.class, args);
    }

    @Override //from CommandLineRunner interface (called automatically)
    public void run(String... args) throws Exception {
        Job job = (Job) applicationContext.getBean("myHelloWorldJob");

        JobParameters jobParameters = new JobParametersBuilder()
        /*Necessary for running several instances of a same job (each jobInstance must have a parameter that changes)*/
        .addLong("timeStampOfJobInstance", System.currentTimeMillis())
        .toJobParameters();
        var jobExecution = jobLauncher.run(job, jobParameters);

        var batchStatus = jobExecution.getStatus();
        while (batchStatus.isRunning()) {
            System.out.println("Job still running...");
            Thread.sleep(5000L);
        }
        System.out.println("Job is finished ...");
    }
}

```

Lancer l'exécution de **TpSpringBatchApplication.main()**

Résultat attendu :

```

...
hello world by SpringBatch
...

```

### Variante (configuration du job en xml) :

Créer le nouveau package "tp.tpSpringBatch.job.xml".

Au sein de ce package créer/coder la nouvelle classe suivante **SomeJobsFromXmlConfig** :

```
package tp.tpSpringBatch.job.xml;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;
import org.springframework.context.annotation.Profile;

@Configuration
@Profile("xmlJobConfig")
@ImportResource({"classpath:job/commonConfig.xml",
                 "classpath:job/myHelloWorldJob.xml"})
public class SomeJobsFromXmlConfig { }
```

Créer le nouveau répertoire "src/main/resources/job".

Au sein de ce répertoire les deux fichiers suivants :

#### *commonConfig.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/batch
                           https://www.springframework.org/schema/batch/spring-batch.xsd
                           http://www.springframework.org/schema/beans
                           https://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- listeners -->
    <!-- readers and writers -->
</beans>
```

#### *myHelloWorldJob.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/batch
                           https://www.springframework.org/schema/batch/spring-batch.xsd
                           http://www.springframework.org/schema/beans
                           https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="printMessageTaskletA" class="tp.tpSpringBatch.tasklet.PrintMessageTasklet">
        <property name="message" value="HELLO WORLD "/>
    </bean>

    <bean id="printMessageTaskletB" class="tp.tpSpringBatch.tasklet.PrintMessageTasklet">
        <property name="message" value="Xml defined Tasklet"/>
    </bean>
```

```
<job id="myHelloWorldJob" xmlns="http://www.springframework.org/schema/batch">

    <step id="step1" next="step2">
        <tasklet ref="printMessageTaskletA" />
    </step>

    <step id="step2" >
        <tasklet ref="printMessageTaskletB" />
    </step>
</job>

</beans>
```

Petit ajustement dans TpSpringBatchApplication.main() pour démarrer le job dans sa variante "configuré en xml" :

```
public static void main(String[] args) {
    String defaultProfils = "xmlJobConfig";
    //String defaultProfils = "";
    System.setProperty("spring.profiles.default", defaultProfils);
    SpringApplication.run(TpSpringBatchApplication.class, args);
}
```

*Résultat attendu :*

```
...
HELLO WORLD
...
Xml defined Tasklet
...
```

**Variante de lancement (via Test JUnit et configuration java) :**

```
tp.tpSpringBatch.job.TestHelloWorldJob (dans src/test/java)
package tp.tpSpringBatch.job;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.test.JobLauncherTestUtils;
import org.springframework.batch.test.context.SpringBatchTest;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.ActiveProfiles;
import tp.tpSpringBatch.TpSpringBatchApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
```

```

import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.Configuration;
import tp.tpSpringBatch.config.AutomaticSpringBootBatchJobRepositoryConfig;

@Configuration
@EnableAutoConfiguration //springBoot & spring-boot-starter-batch autoConfig (application.properties)
@Import({ AutomaticSpringBootBatchJobRepositoryConfig.class,
          HelloWorldJobConfig.class ,
          PrintHelloWorldMessageTaskletBean.class})
class HelloWorldJobTestConfig {

}

@SpringBatchTest
@SpringBootTest(classes = { HelloWorldJobTestConfig.class} )
@ActiveProfiles(profiles = {})
public class TestHelloWorldJob {
    Logger logger = LoggerFactory.getLogger(TestHelloWorldJob.class);

    @Autowired
    private JobLauncherTestUtils jobLauncherTestUtils;

    @Autowired
    //no need of @Qualifier("myHelloWorldJob") because only one unique job should be found
    //in @SpringBatchTest configuration (good practice in V5 , mandatory in SpringBatch V4)
    private Job job;

    @Test
    public void testHelloWorldJob() throws Exception {
        this.jobLauncherTestUtils.setJob(job);
        JobExecution jobExecution = jobLauncherTestUtils.launchJob();
        logger.debug("jobExecution="+jobExecution.toString());
        assertEquals("COMPLETED", jobExecution.getExitStatus().getExitCode());
    }
}

```

Ce test devrait pourvoir être lancé sans échec.

### Variante de lancement (via Test JUnit et configuration des jobs en xml) :

```

tp.tpSpringBatch.job.xml.TestXmlHelloWorldJob (dans src/test/java)
package tp.tpSpringBatch.job;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.test.JobLauncherTestUtils;
import org.springframework.batch.test.context.SpringBatchTest;

```

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.ActiveProfiles;
import tp.tpspringBatch.TpSpringBatchApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.ImportResource;
import org.springframework.context.annotation.Configuration;
import tp.tpspringBatch.config.AutomaticSpringBootBatchJobRepositoryConfig;

@Configuration
@EnableAutoConfiguration //springBoot & spring-boot-starter-batch autoConfig (application.properties)
@Import({AutomaticSpringBootBatchJobRepositoryConfig.class})
@ImportResource({"classpath:job/commonConfig.xml",
                 "classpath:job/myHelloWorldJob.xml"})
class HelloWorldJobXmlTestConfig {

}

@SpringBatchTest
@SpringBootTest(classes = { HelloWorldJobXmlTestConfig.class } )
@ActiveProfiles(profiles = {"xmlJobConfig"})
public class TestXmlHelloWorldJob {
    Logger logger = LoggerFactory.getLogger(TestXmlHelloWorldJob.class);

    @Autowired
    private JobLauncherTestUtils jobLauncherTestUtils;

    @Autowired
    //no need of @Qualifier("myHelloWorldJob") because only one unique job should be found
    //in @SpringBatchTest configuration (good practice in V5 , mandatory in SpringBatch V4)
    private Job job;

    @Test
    public void testHelloWorldJob() throws Exception {
        this.jobLauncherTestUtils.setJob(job);
        JobExecution jobExecution = jobLauncherTestUtils.launchJob();
        logger.debug("jobExecution="+jobExecution.toString());
        assertEquals("COMPLETED", jobExecution.getExitStatus().getExitCode());
    }
}

```

## **Restructuration des classes de tests (via héritage)**

Ajouter les 2 classes abstraites suivantes au sein du package ***tp.tpSpringBatch*** (de la partie src/test) :

### **AbstractBasicTestJobHelper.java**

```
package tp.tpSpringBatch;

import static org.assertj.core.api.Assertions.assertThat; import org.junit.jupiter.api.AfterEach;
import org.slf4j.Logger; import org.slf4j.LoggerFactory;
import org.springframework.batch.core.Job; import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.test.JobLauncherTestUtils;
import org.springframework.batch.test.JobRepositoryTestUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext; import org.springframework.core.io.FileSystemResource;

public abstract class AbstractBasicTestJobHelper {
    protected Logger logger = LoggerFactory.getLogger(AbstractBasicTestJobHelper.class);

    @Autowired
    //no need of @Qualifier("myHelloWorldJob") because only one unique job should be found
    //in @SpringBatchTest configuration (good practice in V5 , mandatory in SpringBatch V4)
    protected Job job;

    @Autowired
    protected ApplicationContext applicationContext;

    @Autowired
    protected JobLauncherTestUtils jobLauncherTestUtils;

    @Autowired
    protected JobRepositoryTestUtils jobRepositoryTestUtils;

    @AfterEach
    public void cleanUp() {
        jobRepositoryTestUtils.removeJobExecutions();
    }

    //to override in subClasses
    public JobParametersBuilder initJobParametersWithBuilder(JobParametersBuilder
jobParametersBuilder) {
        return jobParametersBuilder;
        //return jobParametersBuilder.addString("paramName", "paramValue");
    }

    public JobParameters initJobParameters() {
        JobParametersBuilder jobParametersBuilder = new JobParametersBuilder()
            .addLong("timeStampOfJobInstance", System.currentTimeMillis());
        //Necessary for running several instances of a same job (each job instance must have a parameter that changes)
        jobParametersBuilder = initJobParametersWithBuilder(jobParametersBuilder);
        //for .addString("paramName", "paramValue")
        return jobParametersBuilder.toJobParameters();
    }
}
```

```

public void verifSameContentExceptedResultFile(String expectedFilePath, String actualFilePath){
    FileSystemResource expectedResult = new FileSystemResource(expectedFilePath);
    FileSystemResource actualResult = new FileSystemResource(actualFilePath);
    //AssertFile.assertEquals(expectedResult, actualResult); //deprecated since v5
    assertThat(actualResult.getFile()).hasSameTextualContentAs(expectedResult.getFile());//via AssertJ
    logger.debug(">>> expected_file: " + expectedFilePath
                + " and generated_file: " + actualFilePath + " have same content .");
}

//to override in subclass
public void postJobCheckings() {
    //ex: check generated file or else
}
}

```

### AbstractBasicActiveTestJob.java

```

package tp.tpSpringBatch;
import static org.junit.jupiter.api.Assertions.assertEquals; import org.junit.jupiter.api.Test;
import org.springframework.batch.core.ExitStatus; import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.JobInstance; import org.springframework.batch.core.JobParameters;

public abstract class AbstractBasicActiveTestJob extends AbstractBasicTestJobHelper{

    @Test
    public void basicGenericTestJob() throws Exception {
        JobParameters jobParameters = initJobParameters();
        logger.debug(">>> jobName=" + job.getName());
        JobExecution jobExecution = jobLauncherTestUtils.launchJob(jobParameters);
        logger.debug("jobExecution=" + jobExecution.toString());

        JobInstance actualJobInstance = jobExecution.getJobInstance();
        assertEquals(job.getName(), actualJobInstance.getJobName());

        ExitStatus actualJobExitStatus = jobExecution.getExitStatus();
        assertEquals("COMPLETED", actualJobExitStatus.getExitCode());

        postJobCheckings();
    }
}

```

Ceci permet de retoucher (par simplification) les classes de nos tests :

```

...
public class TestHelloWorldJob extends AbstractBasicActiveTestJob{
}

```

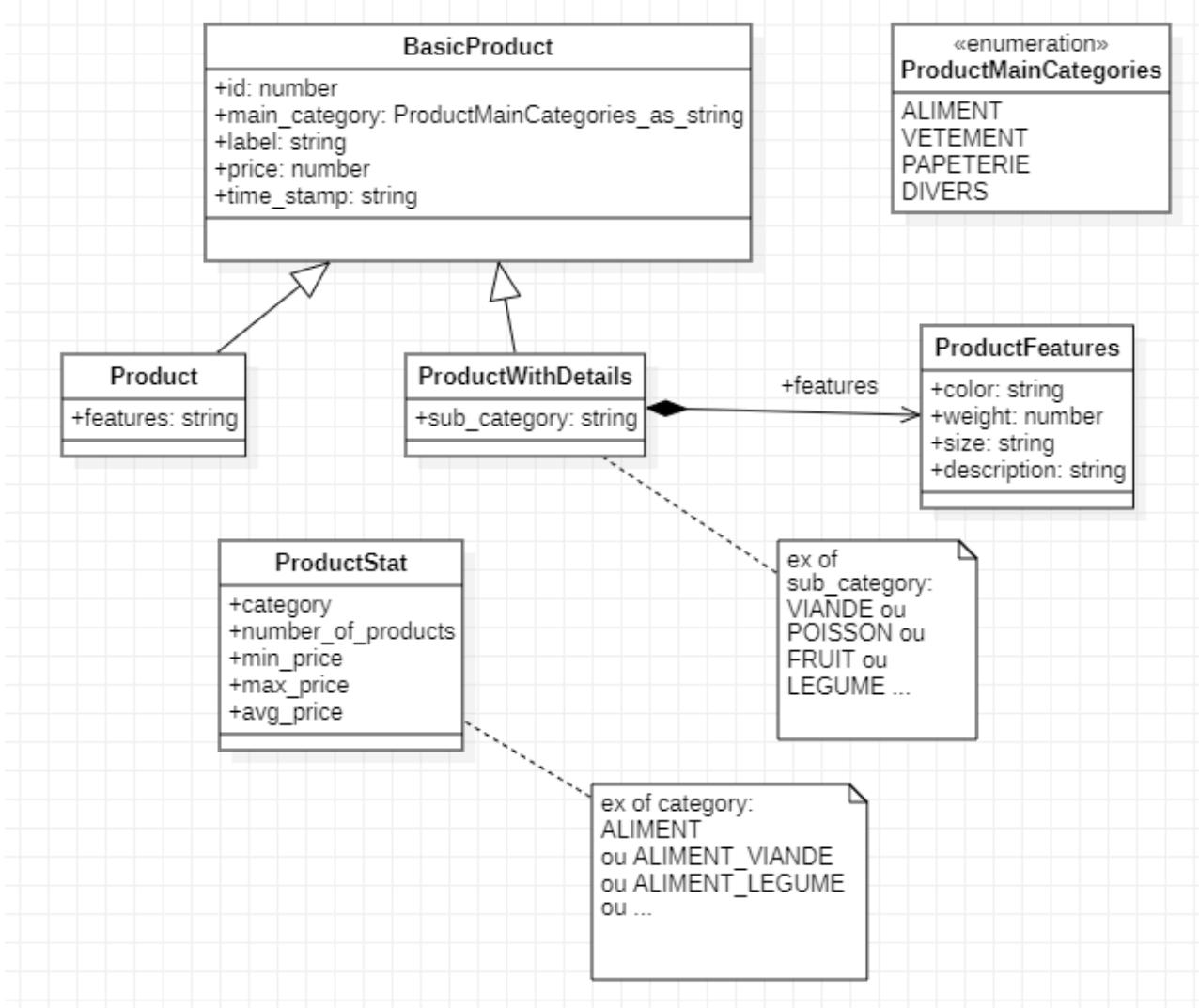
```

...
public class TestXmlHelloWorldJob extends AbstractBasicActiveTestJob{
}

```

Dans la suite des TPs ci après, on pourra choisir une configuration "**java**" ou bien "**xml**" ou bien "**java puis xml**" des "**step**" de des "**job**" de springBatch ...

### 1.3. Structures de données pour les Tps



#### Première structuration du projet

dans un nouveau package **tp.tpSpringBatch.model** coder les classes **BasicProduct** et **Product** selon le diagramme UML ci dessus.

NB : pour gagner un peu de temps , on pourra éventuellement s'inspirer du projet "tpSpringBatch" (en phase "début de tp") disponible dans la partie "**tp**" du référentiel  
<https://github.com/didier-tp/tp-spring-batch.git>

A la racine du projet (à coté de src) préparer l'arborescence de répertoires suivante :

**data/input/csv**

**data/output/json**

**data/expected\_output/json**

*Exemple de fichier à traiter : data/input/csv/**products.csv***

id;main_category;label;price;time_stamp;features
1;aliment;pommes;2.2;2024-05-23T11:31:00;golden
2;aliment;bananes;1.8;2024-05-23T11:31:00;cameroun
3;vetement;t-shirt;7.7;2024-05-23T11:31:00;blanc xl
4;vetement;pull;12.75;2024-05-23T11:31:00;beige xl

NB : on pourra éventuellement/facultativement s'appuyer sur lombok

## 1.4. Csv to console puis Csv to Json

Réglage préliminaire :

ajouter ceci dans src/main/resources/**application.properties**

```
#disable auto launching of jobs (spring-boot-starter-batch)
```

```
spring.batch.job.enabled=false
```

```
#if not .enabled=false the property spring.batch.job.name must be set in case of multiple jobs
```

*Car sans ce réglage "springBoot" et "spring-boot-starter-batch" tentent par défaut un démarrage (très/trop automatique) de job .*

Phase1 du Tp :

Ecrire et lancer un job de nom "**fromCsvToConsoleJob**" qui va lire un fichier **products.csv** et générer un **affichage à la console** .

On pourra s'appuyer sur un chunk utilisant un reader de type "csv" et un writer basé sur une instance la nouvelle classe suivante :

**tp.tpSpringBatch.writer.custom.SimpleObjectWriter**

```
package tp.tpSpringBatch.writer.custom;
import org.springframework.batch.item.Chunk;
import org.springframework.batch.item.ItemWriter;

public class SimpleObjectWriter<T extends Object> implements ItemWriter<T> {

    public SimpleObjectWriter() {
    }

    @Override
    public void write(Chunk<? extends T> chunk) throws Exception {
        for(T obj : chunk) {
            System.out.println(obj.toString());
        }
    }
}
```

NB: configuration (java et/ou xml) et lancement (main et/ou JUnit) libres (selon envies et temps).

*Exemple partiel d'affichage console à peu près attendu (selon code de Product.toString()) :*

```
Product [features=golden] BasicProduct(id=1, main_category=aliment, label=pommes, price=2.2, time_stamp=2024-05-23T11:31:00)
Product [features=beige xl] BasicProduct(id=4, main_category=vetement, label=pull, price=12.75, time_stamp=2024-05-23T11:31:00)
```

Phase 2 du Tp :

Adapter le job précédent "**fromCsvToConsoleJob**" de façon à déclencher un "**processeur**" qui va transformer la partie **.main\_category** des produits en majuscules .

*Exemple partiel d'affichage console à peu près attendu (selon code de Product.toString()) :*

```
Product [features=golden] BasicProduct(id=1, main_category=ALIMENT, label=pommes, price=2.2, time_stamp=2024-05-23T11:31:00)
Product [features=beige xl] BasicProduct(id=4, main_category=VETEMENT, label=pull, price=12.75, time_stamp=2024-05-23T11:31:00)
```

### Phase 3 du Tp :

En s'inspirant du job précédent créer un nouveau job de nom "**fromCsvToJsonJob**" de façon à générer un fichier data/output/json/**products.json** en sortie :

Nouvelle dépendances nécessaire pour traiter le format "json" (dans **pom.xml**):

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
</dependency>
```

Exemple de fichier attendu en sortie :

```
[{"id":1,"main_category":"ALIMENT","label":"pommes","price":2.2,"time_stamp":"2024-05-23T11:31:00","features":"golden"}, {"id":2,"main_category":"ALIMENT","label":"bananes","price":1.8,"time_stamp":"2024-05-23T11:31:00","features":"cameroun"}, {"id":3,"main_category":"VETEMENT","label":"t-shirt","price":7.7,"time_stamp":"2024-05-23T11:31:00","features":"blanc xl"}, {"id":4,"main_category":"VETEMENT","label":"pull","price":12.75,"time_stamp":"2024-05-23T11:31:00","features":"beige xl"}]
```

## 1.5. Csv to Xml , ... , Job avec paramètres

### Phase 1 du Tp :

Ecrire et lancer un job de nom "**fromCsvToXmlJob**" qui va lire un fichier **products.csv** et générer un fichier **products.xml**.

Certaines dépendances complémentaires seront à priori nécessaires au sein de **pom.xml** :

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-oxm</artifactId>
</dependency>
<dependency>
    <groupId>com.thoughtworks.xstream</groupId>
    <artifactId>xstream</artifactId>
    <version>1.4.20</version>
</dependency>
<dependency>
    <groupId>jakarta.xml.bind</groupId>
    <artifactId>jakarta.xml.bind-api</artifactId>
</dependency>
<dependency>
    <groupId>org.glassfish.jaxb</groupId>
    <artifactId>jaxb-runtime</artifactId>
</dependency>
```

Il faudra peut être ajouter cette annotation au dessus de la classe Product :

```
@XmlElement(name = "product") //just for read/generate XML file with jaxb2 marshaller
```

**exemple de fichier généré data/output/xml/products.xml**

```
<?xml version="1.0" encoding="UTF-8"?><products>
<product><id>1</id><label>pommes</label><main_category>ALIMENT</main_category><price>2.2</price>
<time_stamp>2024-05-23T11:31:00</time_stamp><features>golden</features></product>
<product><id>2</id><label>bananes</label><main_category>ALIMENT</main_category><price>1.8</price>
<time_stamp>2024-05-23T11:31:00</time_stamp><features>cameroun</features></product>
<product><id>3</id><label>t-shirt</label><main_category>VETEMENT</main_category><price>7.7</price>
<time_stamp>2024-05-23T11:31:00</time_stamp><features>blanc xl</features></product>
<product><id>4</id><label>pull</label><main_category>VETEMENT</main_category><price>12.75</price>
<time_stamp>2024-05-23T11:31:00</time_stamp><features>beige xl</features></product>
</products>
```

**Phase 2 du Tp :**

Les chemins menant aux fichiers à lire et écrire seront des **paramètres variables du job**.

Noms et valeurs des paramètres du job :

<b>inputFilePath</b>	"data/input/csv/products.csv"
<b>outputFilePath</b>	"data/output/xml/products.xml"
<b>enableUpperCase</b>	true

**NB :** au sein d'une configuration java , on pourra s'inspirer de ce code :

```
@Bean(destroyMethod="")
@Qualifier("xml")
@StepScope
ItemStreamWriter<Product> productXmlFileItemWriter(
    @Value("${jobParameters['outputFilePath']}") String outputPath
) {
    //logger.info("in @Bean productXmlFileItemWriter() , outputPath="+outputPath);
    WritableResource outputXmlResource = new FileSystemResource(outputPath);
    ...
}
```

et au sein d'une configuration xml on pourra s'inspirer de ce code :

```
<bean id="productXmlFileItemWriter"
class="org.springframework.batch.item.xml.StaxEventItemWriter" scope="step" >
    <!-- <property name="resource" value="file:data/output/xml/products.xml" /> -->
    <property name="resource" value="#{jobParameters['outputFilePath']}" />
    ...

```

Selon la version utilisée de SpringBatch, on aura peut être besoin de ce complément de configuration dans **application.properties** (pour accompagner une configuration XML) :

```
#temp workaround for step scope in job xml config file
#a priori (probleme interpretation selon namespace xml)
# (https://github.com/spring-projects/spring-batch/issues/3936 ) pas clair ...
spring.main.allow-bean-definition-overriding=true
```

NB :

On pourra éventuellement lancer l'application SpringBatch via un script de ce genre (à adapter) :

***lancer\_fromCsvToXmlJob.bat***

```
set PROJECT_ROOT=..\..\..  
set CP=target\tpSpringBatchSolution-0.0.1-SNAPSHOT.jar  
REM set MAIN_CLASS=tp.tpSpringBatch.TpSpringBatchApplication  
set JOB_NAME=fromCsvToXmlJob  
set inputFilePath=data/input/csv/products.csv  
set outputPath=data/output/xml/products.xml  
  
REM set PATH="C:\Program Files\Java\jdk-17\bin"  
cd /d %~dp0%PROJECT_ROOT%  
  
REM java -Dspring.profiles.active=xmlJobConfig ...-DjobName=%JOB_NAME% -jar %CP%  
java -DinputFilePath=%inputFilePath% -DoutputFilePath=%outputFilePath%  
-jar %CP% %JOB_NAME%  
pause
```

## 1.6. Avec lectures et insertions en base relationnelle

### 1) Lancer quelques scripts SQL pour préparer la base de données "*productdb*"

#### **src/main/resources/sql/batch-schema.sql**

```
DROP TABLE IF EXISTS product_with_details;
```

```
CREATE TABLE product_with_details (
```

```
    id BIGINT AUTO_INCREMENT NOT NULL PRIMARY KEY,
    main_category VARCHAR(50),
    sub_category VARCHAR(50),
    label VARCHAR(50),
    price double,
    time_stamp VARCHAR(30),
    f_color VARCHAR(32),
    f_weight double,
    f_size VARCHAR(16),
    f_description VARCHAR(128)
);
```

#### **src/main/resources/sql/insert-products.sql**

```
INSERT INTO product_with_details (main_category,sub_category,label,price,time_stamp,f_color,f_weight,f_size,f_description)
VALUES('aliment','fruit','pommes',2.2,'2024-05-23T11:31:00','rouge',1500,'1l','golden');
INSERT INTO product_with_details (main_category,sub_category,label,price,time_stamp,f_color,f_weight,f_size,f_description)
VALUES('aliment','fruit','bananes',1.8,'2024-05-23T11:31:00','jaune',1100,'1l','cameroun');
INSERT INTO product_with_details (main_category,sub_category,label,price,time_stamp,f_color,f_weight,f_size,f_description)
VALUES('vetement',null,'t-shirt',7.7,'2024-05-23T11:31:00','blanc',200,'xl','en coton');
INSERT INTO product_with_details (main_category,sub_category,label,price,time_stamp,f_color,f_weight,f_size,f_description)
VALUES('vetement',null,'pull',12.75,'2024-05-23T11:31:00','beige',400,'xl','en laine');
```

#### **src/main/resources/sql/select-products.sql**

```
SELECT id,main_category,sub_category,label,price,time_stamp,f_color,f_weight,f_size,f_description FROM product_with_details;
```

**Les scripts précédents (en syntaxe "h2") pourront éventuellement être copié/collés et lancés depuis une console H2 que l'on pourra éventuellement lancer via ces scripts :**

#### **src/script/h2/*set-env.bat***

```
set MVN_REPOSITORY=C:\Users\formation\.m2\repository
set MY_H2_DB_URL_PRODUCT=jdbc:h2:~/productDb
set MY_H2_DB_URL_JOBREPOSITORY=jdbc:h2:~/jobRepositoryDb

set PATH="C:\Prog\java\eclipse-jee-2023-12\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.9.v20231028-0858\jre\bin"

set H2_VERSION=2.2.224
set H2_CLASSPATH=%MVN_REPOSITORY%\com\h2database\h2%H2_VERSION%\h2-%H2_VERSION%.jar
```

#### **src/script/h2/lancer\_console\_h2\_productdb.bat**

```
cd /d %~dp0
call set_env.bat
java -jar %H2_CLASSPATH% -user "sa" -url %MY_H2_DB_URL_PRODUCT%
```

*REM NB: penser à se déconnecter pour éviter des futurs verrous/blocages*  
pause

## 2) Configuration d'un accès à la base de données depuis l'application "springBatch" :

ajouter ceci dans src/main/resources/**application.properties**

```
#secondary DataBases for some Jobs:
spring.productdb.datasource.url=jdbc:h2:~/productDb
spring.productdb.datasource.driverClassName=org.h2.Driver
spring.productdb.datasource.username=sa
spring.productdb.datasource.password=
```

Ajouter cette classe **tp.tpSpringBatch.datasource.MyProductDbDataSourceConfig**

```
package tp.tpSpringBatch.datasource;

import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.autoconfigure.jdbc.DataSourceProperties;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MyProductDbDataSourceConfig {

    @Bean @Qualifier("productdb")
    @ConfigurationProperties("spring.productdb.datasource")
    public DataSourceProperties productdbDataSourceProperties() {
        return new DataSourceProperties();
    }

    @Bean(name="productdbDataSource") @Qualifier("productdb")
    public DataSource productdbDataSource(@Qualifier("productdb") DataSourceProperties
productdbDataSourceProperties) {
        return productdbDataSourceProperties
            .initializeDataSourceBuilder()
            .build();
    }
}
```

## 3) Coder et lancer un job augmentant le prix de tous les produits de 1 %

*Indications pour un code en pur java (sans xml):*

\* coder la classe tp.tpSpringBatch.db.**ProductWithDetailsRowMapper** en prenant en compte le fait que toutes les colonnes *f\_...* sont à charger dans un sous objet "features" / ProductFeatures .

\* coder la classe tp.tpSpringBatch.reader.java.**MyDbProductWithDetailsReaderConfig** déclenchant une requête SELECT et en s'appuyant sur le RowMapper précédent.

\* coder la classe tp.tpSpringBatch.processor.**IncreasePriceOfProductWithDetailsProcessor** augmentant le prix des produits et actualisant idéalement le time\_stamp .

\* coder la classe `tp.tpSpringBatch.writer.java.MyConsoleProductWithDetailsWriterConfig` pour un premier test avec un affichage dans la console .

\* coder une première version de la classe `tp.tpSpringBatch.job.java.IncreaseProductPriceInDbJobConfig` de manière à lire des produits en base , augmenter le prix via le processeur et effectuant un affichage console via le writer temporaire

\* coder et lancer une classe de test (ex : `tp.tpSpringBatch.job.TestIncreaseProductPriceInDbJob`)

\* coder la classe `tp.tpSpringBatch.writer.java.MyDbProductWithDetailsWriterConfig` déclenchant une requête SQL de ce type "`UPDATE product_with_details SET price = :price , time_stamp = :time_stamp WHERE id = :id`" au sein d'une méthode de ce genre :

```
@Bean @Qualifier("update_price_in_db")
public ItemWriter<ProductWithDetails>
updatePriceOfProductWithDetailsJdbcItemWriter(@Qualifier("productdb") DataSource
productdbDataSource) {...}
```

\* ajuster le code de la classe `tp.tpSpringBatch.job.java.IncreaseProductPriceInDbJobConfig` de manière à y remplacer le writer ("console" → "update\_price\_in\_db" )

\* ajuster et relancer la classe de test (ex: `tp.tpSpringBatch.job.TestIncreaseProductPriceInDbJob`)

NB : on pourra éventuellement préférer un code "java + XML" pour la configuration du job . Variantes (avec parties configurées en xml) :

- `ProductWithDetailsRowMapper` et `IncreasePriceOfProductWithDetailsProcessor` comme en config java

- placer la config xml du `jdbcProductWithDetailsReader` au sein de `job/rw/jdbcReader.xml`

- on pourra réutiliser le `productConsoleItemWriter` basé sur `tp.tpSpringBatch.writer.custom.SimpleObjectWriter` et configuré dans `job/rw/consoleWriter.xml` pour le premier test préliminaire avec écriture à la console

- coder une première version du job `increaseProductPriceInDbJob` au sein de `job/increaseProductPriceInDbJob.xml` s'appuyant sur `jdbcProductWithDetailsReader` , `increasePriceOfProductWithDetailsProcessor` et `productConsoleItemWriter` .

- coder une première version de `TestXmlIncreaseProductPriceInDbJob` en ajustant bien les dépendances java et xml nécessaires (ex : `classpath:job/jdbcCommonSubConfig.xml`)

- coder le `updatePriceOfProductWithDetailsJdbcItemWriter` au sein de `job/rw/jdbcWriter.xml`

- ajuster le code de `job/increaseProductPriceInDbJob.xml` et le `test` qui va avec pour vérifier le bon fonctionnement du Job avec écriture en base des prix modifiés

## 1.7. Avec flow conditionnel basé sur JobExecutionDecider

1) ajuster le code de `IncreasePriceOfProductWithDetailsProcessor` de manière à ce que seuls les prix des produits d'une certaine catégorie soient augmentés tout en stockant le nombre de prix modifiés dans l'attribut "nbAdjustedProducts" de `stepExecution.getExecutionContext()` .

On pourra pour cela s'appuyer sur les paramètres variables suivant du processeur (avec `@StepScope`) :

```
@Value("#{jobParameters['increaseRatePct']}")  
private Double increaseRatePct = 0.0;//taux d'augmentation en %  
  
@Value("#{jobParameters['productCategoryToIncrease']}")
```

```

private String productCategoryToIncrease = "?"; //ex: "aliment" ou "vetement" ou ...
@Value("#{stepExecution}") //ok with @StepScope
private StepExecution stepExecution;
    
```

**2) coder la classe *MyUpdatedCountCheckingDecider*** (implémentant l'interface *JobExecutionDecider*) qui va analyser la valeur de l'attribut "nbAjustedProducts" de stepExecution.getExecutionContext() et qui va retourner FlowExecutionStatus("COMPLETED\_WITH\_MANY\_UPDATED") que si la valeur de nbAjustedProducts est supérieure ou égale à la valeur de jobExecution.getJobParameters().getLong("minManyUpdated").

**3) améliorer le code de *IncreaseProductPriceInDbJobConfig*** en utilisant une instance du "decider" précédent de manière à enchaîner alternativement une de ces deux "step" selon la valeur "COMPLETED" ou "COMPLETED\_WITH\_MANY\_UPDATED" du "decider" :

```

Step stepWhenFew = stepBuilder.tasklet(
    new PrintMessageTasklet(">>>> few updated products"), this.batchTxManager).build();
    
```

```

Step stepWhenMany = stepBuilder.tasklet(
    new PrintMessageTasklet(">>>> MANY updated products"), this.batchTxManager).build();
    
```

**4) améliorer le test unitaire *TestIncreaseProductPriceInDbJob*** en fixant certains paramètres via un code de ce genre :

```

.addDouble("increaseRatePct", 1.0) //used by IncreasePriceOfProductWithDetailsProcessor (1% d'augmentation)
.addString("productCategoryToIncrease", "aliment") //used by IncreasePriceOfProductWithDetailsProcessor
                                                // (catégorie de produit à augmenter)
.addLong("minManyUpdated", 2L); //used by MyUpdatedCountCheckingDecider
    
```

Effet à constater à la console :

>>>> MANY updated products  
ou bien

>>>> MANY updated products

**5) partie facultative du TP :** améliorer et tester le code de *IncreaseProductPriceInDbJobConfig* de manière à générer en plus un fichier de statistiques si "COMPLETED\_WITH\_MANY\_UPDATED" est retourné par le "decider".

Indication : on pourra se baser sur les éléments suivants (à coder) :

db.ProductStatRowMapper	implements RowMapper<ProductStat>
MyDbProductStatReaderConfig	Déclenchant cette requête SQL : $  \begin{aligned}  &\text{SELECT main\_category as category,} \\  &\text{count(*) as nb\_prod, min(price) as min\_price,} \\  &\text{max(price) as max\_price, avg(price) as avg\_price} \\  &\text{FROM PRODUCT\_WITH\_DETAILS} \\  &\text{GROUP BY main\_category}  \end{aligned}  $
MyCsvFileProductStatWriterConfig	Générant un fichier <b>productStats.csv</b>

Effet à constater dans data/output/csv :

fichier **productStats.csv** généré ou pas

category;number_of_products;min_price;max_price;avg_price
aliment;2;1.9298436337926181;2.3586977746354223;2.14427070421402
vetement;2;7.777;12.8775;10.32725

## 1.8. TP facultatif: Exécution en parallèle (partitions)

Coder et tester une version améliorée de **IncreaseProductPriceInDbJobConfig** nommée **IncreaseProductPriceInDbJobWithPartitionConfig** au sein de laquelle on répartira les traitements au sein de plusieurs partitions (exécutées par différents Threads).

## 1.9. csv (éventuellement hybride) vers DataBase avec gestion des incidents (retry ou restart) .

- 1) Coder et tester un job qui va lire des données dans un fichier **data/input/csv/newProducts.csv** et les insérer dans la table "**product\_with\_details**" de la base "**productdb**" .
- 2) on codera une variante erronée du fichier d'entrée **data/input/csv/newProductsWithorWithoutErrors.csv** de manière à tester le comportement de "retry" et "restart" .

## 1.10. Monitoring de job

Réfléchir à une stratégie permettant de surveiller l'exécution de certains Jobs .  
Mettre un embryon de cela en place si le temps et les moyens à disposition le permettent.