

Spring Batch

Table des matières

I - Spring Batch (vue d'ensemble , démarrage).....	4
1. Traitements de masse par lots.....	4
1.1. Besoins techniques liés aux traitements de masse.....	4
2. Spring Batch (architecture , vue d'ensemble).....	4
2.1. Structuration en couches et abstraction.....	5
2.2. lecture unitaire , écriture multiple (par bloc):.....	6
2.3. Eléments de spring-batch:.....	7
2.4. Distinction entre "Job" , "JobInstance" et "JobExecution":.....	8
2.5. "Step" et "Step Execution".....	9
3. Evolution de Spring Batch (historique).....	10
4. Configuration maven sans springBoot.....	11
5. Configuration maven moderne (avec springBoot).....	12
6. Configurer et Lancer un "Job" simple.....	13
6.1. Rappel du contexte/cadre d'une application springBatch :.....	13
6.2. Configuration explicite: "JobRepository" et "JobLauncher".....	13
6.3. Configuration implicite de springBatch via springBoot.....	17
6.4. Configuration minimaliste d'un tasklet et d'un job en java.....	19
6.5. Lancement d'un job depuis java en ligne de commande.....	21
6.6. Configuration d'un job en xml.....	23
6.7. Lancement d'un job depuis un test JUnit.....	26

II - Tasklet et Chunk (CSV, XML, JSON, ...).....28

1. Tasklets et chunks.....	28
1.1. Exemples d'utilisation de tasklets :.....	28
1.2. Exemples d'utilisation des chunks.....	28
2. Orientation "chunk".....	28
2.1. Traitement par blocs ("chunks").....	28
2.2. "ItemProcessor" simple.....	29
3. Lecture/écriture de fichiers avec spring-batch.....	30
3.1. Gestion des fichiers plats (.txt , .csv , ...).....	30
3.2. Gestion des .csv (avec enregistrements délimités (";")).....	31
3.3. Gestion des fichiers plats à positions fixes.....	38
3.4. Gestions de fichiers ".csv / .txt" avec format Hybride.....	43
3.5. Gestion des fichiers json.....	46
3.6. Gestion des fichiers XML.....	51

III - Job Parameters, Chunk jdbc , flows ,57

1. Job avec paramètres.....	57
1.1. Lancement d'un job avec des paramètres.....	57
1.2. Prise en compte des paramètres d'un job.....	58
1.3. Validation et utilisation des "jobParameters".....	59
1.4. 2 jobInstances must at least have a different jobParameter.....	60
2. Processeurs (filtrage, composition, ...).....	61
2.1. Filtrage de données via un processeur.....	61
2.2. Enchaînement de processeurs.....	61
3. Gestion des bases de données.....	62
4. "Step flow" / configuration.....	66
4.1. Enchaînements séquentiels.....	66
4.2. Enchaînements conditionnels simples.....	67
4.3. Contrôle du "ExitStatus" via un Listener.....	69
4.4. Enchaînements conditionnels avec "decider".....	69
4.5. Quelques autres possibilités.....	72
4.6. Enchaînements en parallèle via des partitions.....	73

IV - Tests, Reprise sur erreurs , Monitoring ,81

1. Tests unitaires (pour batch).....	81
1.1. Test unitaire pour job.....	81
1.2. Test unitaire pour "step" individuel.....	83
1.3. Validateurs/assertions classiques et classe abstraite.....	84
2. Reprises sur erreurs.....	86
2.1. Skip some errors.....	86
2.2. Retry when errors in step.....	88

2.3. Restart of job/batch.....	90
3. "ItemReaders" , "ItemWriters" personnalisés.....	95
3.1. "ItemReader" personnalisé :générer jeux de données.....	95
3.2. "ItemWriter" personnalisé (écritures multiples).....	96
4. Lancement de batch.....	97
4.1. Via un "scheduler".....	97
4.2. Via Spring_cloud_data_flow.....	97
5. Monitoring de job.....	98
5.1. Contexte général.....	98
5.2. Exemples (ancients et récents).....	98
5.3. Via Spring_Cloud_DataFlow.....	98
6. Essentiel sur "Spring_Cloud_DataFlow".....	99
6.1. Installation de "Spring_cloud_DataFlow".....	99
6.2. Prise en charge d'une application "springBatch".....	99
6.3. Démarrage et supervision.....	100
7. Aspects divers et avancés.....	103
7.1. logging (pour batch).....	103
7.2. Principaux patterns (pour batch).....	103

V - Annexe – TP Spring Batch..... 105

1. TP - spring-batch.....	105
1.1. Configuration d'un projet spring-batch.....	105
1.2. Programmation et démarrage d'un job très simple.....	105
1.3. Structures de données pour les Tps.....	115
1.4. Csv to console puis Csv to Json.....	116
1.5. Csv to Xml , ... , Job avec paramètres.....	117
1.6. Avec lectures et insertions en base relationnelle.....	120
1.7. Avec flow conditionnel basé sur JobExecutionDecider.....	122
1.8. TP facultatif: Exécution en parallèle (partitions).....	124
1.9. csv (éventuellement hybride) vers DataBase avec gestion des incidents (retry ou restart)	124
1.10. Monitoring de job.....	124

I - Spring Batch (vue d'ensemble , démarrage)

1. Traitements de masse par lots

Bien qu'il existe des technologies de synchronisation efficaces entre applications en quasi temps réel telles que kafka ou autres , on a encore quelquefois besoin d'effectuer en tâche de fond des gros traitements de masse pour l'une de ces raisons :

- transférer des données d'un référentiel à un autre.
- resynchroniser des modifications pour l'instant locales
- vérifier l'exactitude de certaines données (encore à jour , cohérentes , pas obsolètes , ...)
- effectuer des migrations de données (versions et formats éventuellement différents)
- effectuer des extractions , des statistiques , des consolidations/regroupements , ...
- gérer des sauvegardes de données
- ...

"Batch Processing" = "Répétition de traitements sur des grands ensembles de données"

1.1. Besoins techniques liés aux traitements de masse

- **Sans intervention d'un utilisateur (pas d'interface graphique/IHM/GUI)**
- Efficacité / performances (traitements lourds en usage CPU/disques et longs (plusieurs minutes ou plusieurs heures) à optimiser) .
- Supervision : on doit savoir ce qui a été effectué (déjà fait, reste à faire) et savoir si cela s'est bien passé (sans erreur, ...)
- fiabilité : possibilité de reprise d'un paquet de traitements qui ont échoué (transaction par lot)
- Un batch n'est pas un "scheduler" mais un "scheduler" (tel que "quartz") peut le lancer .
- ...

2. Spring Batch (architecture , vue d'ensemble)

"Spring Batch" est un framework additionnel (une **extension** vis à vis de "*Spring framework*").

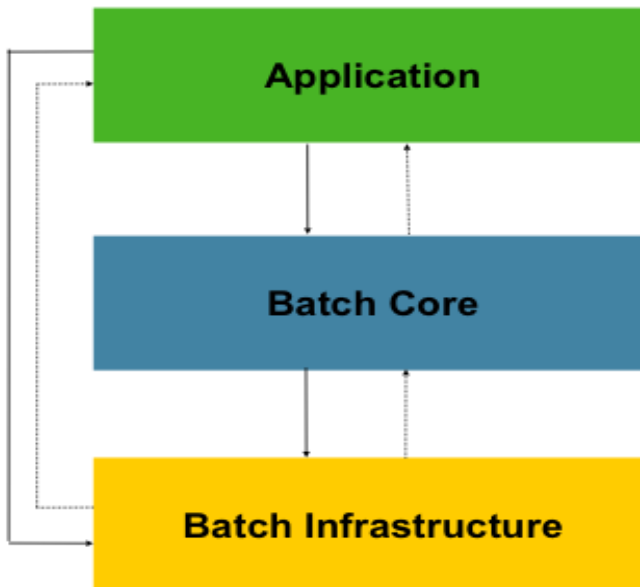
Ce framework est **spécialisé dans la gestion des traitements "batch"** .

Les principales fonctionnalités apportées par "Spring Batch" sont les suivantes:

- Programmation en java et paramétrages souples (anciennement xml , récemment via .properties ou .yaml) .
- Écritures par gros bloc d'enregistrements (pour efficacité/performance)
- Suivi (éventuellement stocké en base ou autre) du déroulement d'un batch .

- Support de plusieurs technologies (fichiers plats (.csv , ...) , fichiers xml , base de données , ...).

2.1. Structuration en couches et abstraction



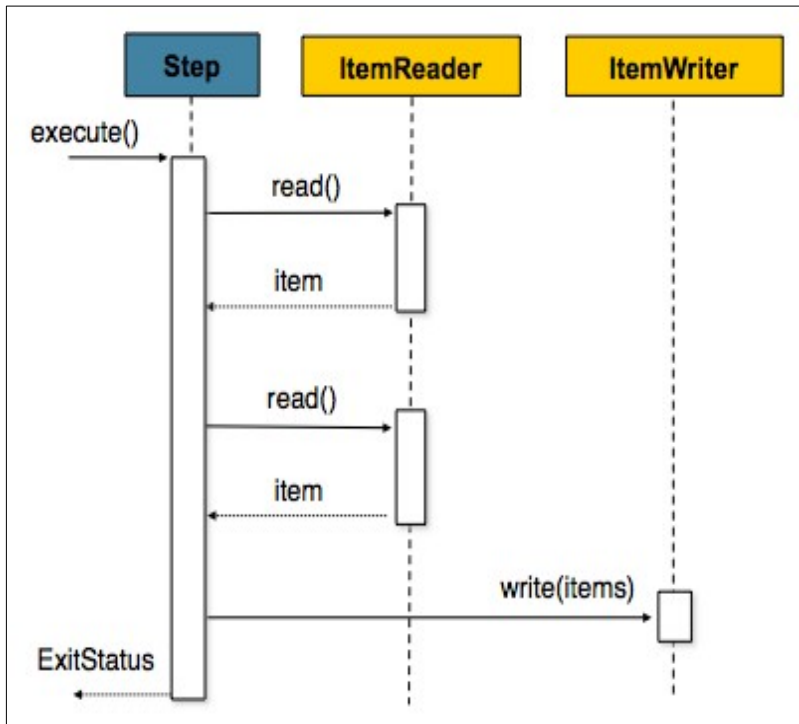
La couche "*Application*" correspond au code à écrire (lié au fonctionnel) .

La couche "*Batch Core*" correspond à l'essentiel du framework "Spring batch" (sa logique , ses automatismes ,)

La couche "*Batch Infrastructure*" correspond à l'infrastructure technique sous sous-jacente (accès aux fichiers , aux bases de données , gestion des transactions , ...).

L'essentiel de "Spring batch" réside en une abstraction suffisamment souple et universelle (au niveau de la couche du milieu "Spring Batch Core") pour s'adapter à différentes technologies et pour être utile à différentes applications (au fonctionnels spécifiques) .

2.2. lecture unitaire , écriture multiple (par bloc):



Terme anglais souvent utilisé par Spring batch :

chunk = gros morceau (gros bloc d'enregistrements) .

```

List items = new ArrayList();
for(int i = 0; i < commitInterval; i++){
    items.add(itemReader.read());
}
itemWriter.write(items);
  
```

```

public interface ItemReader<T> {
    T read() throws Exception, UnexpectedInputException, ParseException;
}
  
```

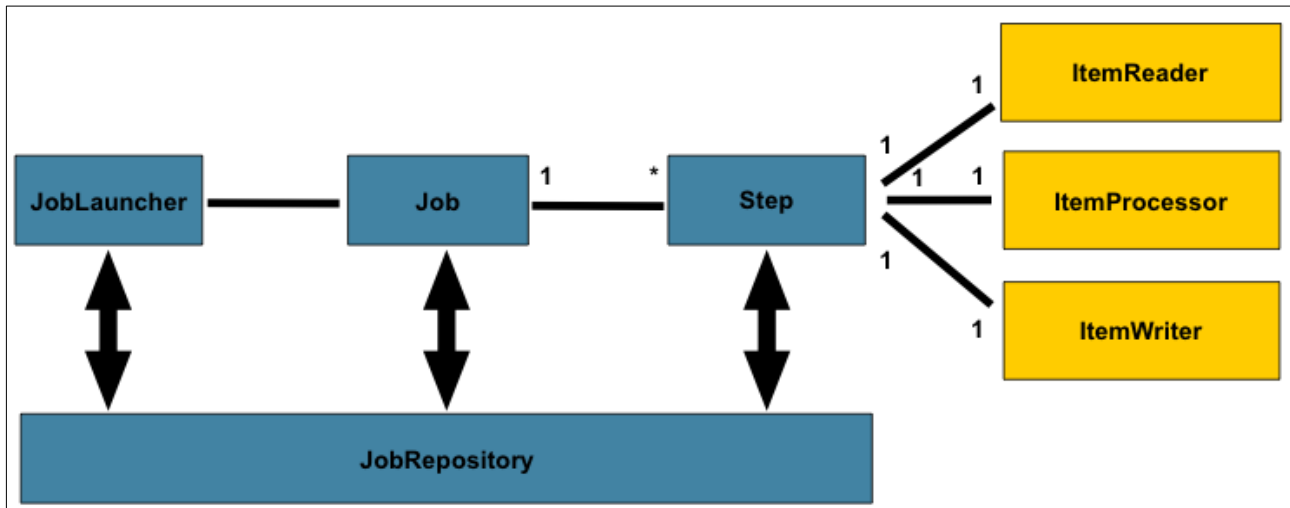
```

public interface ItemWriter<T> {
    //void write(List<? extends T> items) throws Exception; //in old SpringBatch versions
    void write(Chunk<? extends T> items) throws Exception;
}
  
```

NB : La classe **Chunk**<W> du package org.springframework.batch.core.step.item

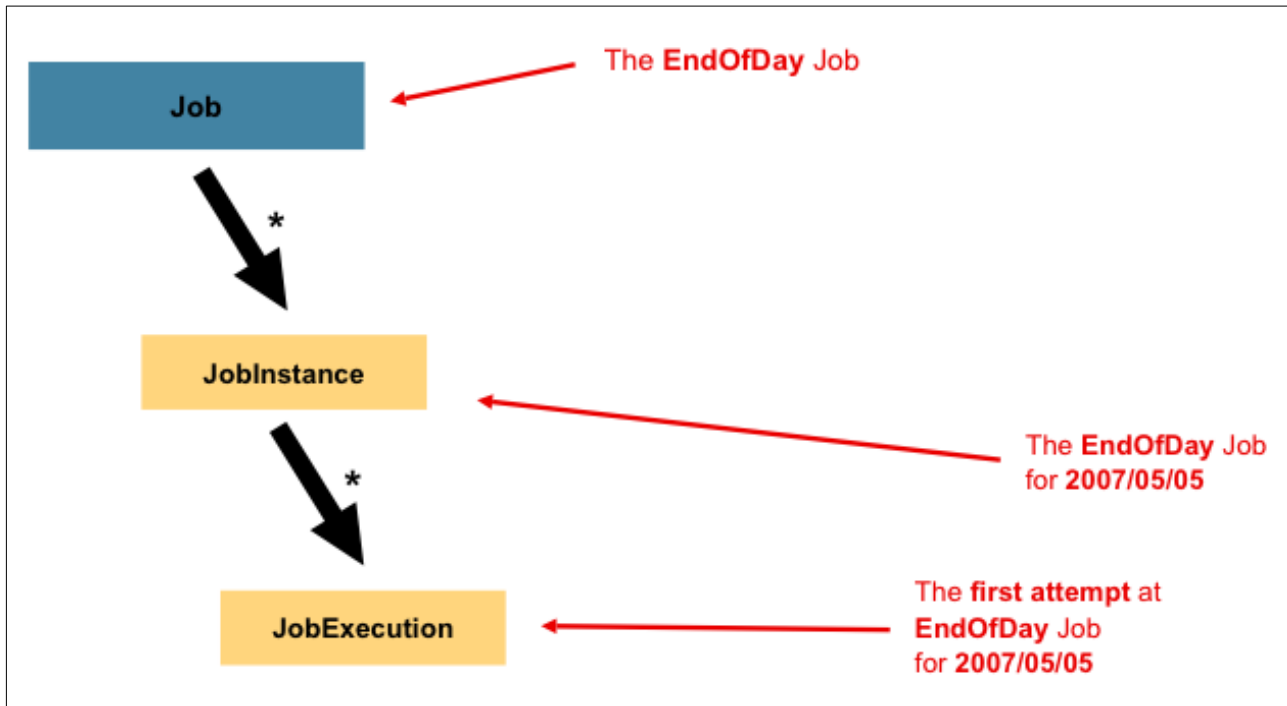
est une encapsulation d'une liste d'items à écrire (List<W>) et d'une liste d'items en échec (à ne pas écrire) .

2.3. Éléments de spring-batch:



ItemReader	Objet spécialisé dans la lecture d'un enregistrement
ItemProcessor	Objet de traitement (un enregistrement en entrée , un autre en sortie)
ItemWriter	Objet permettant d'enregistrer un paquet d'enregistrements
Step	étape d'une tâche (d'un "Job") avec éventuelle étape suivante .
Tasklet (facultatif)	Tâche unique à effectuer au sein d'un "step" ça peut servir à déclencher une procédure stockée ou bien à effectuer des tâches quelconques (hello-world , ...)
Job	Tâche à accomplir en mode "batch" / différé .
JobLauncher	Lanceur de "tâche"/"job"
JobRepository	Référentiel (de données de suivi) pour les tâches en mode batch . Ce référentiel pouvait (dans les versions antérieures) être en mode "Map" ou "DataBase". Aujourd'hui (en mode DB imposé) on peut utiliser une persistance simple (in-memory embeddedDB telle que h2:mem).
ExecutionContext	Contexte d'exécution .

2.4. Distinction entre "Job" , "JobInstance" et "JobExecution":

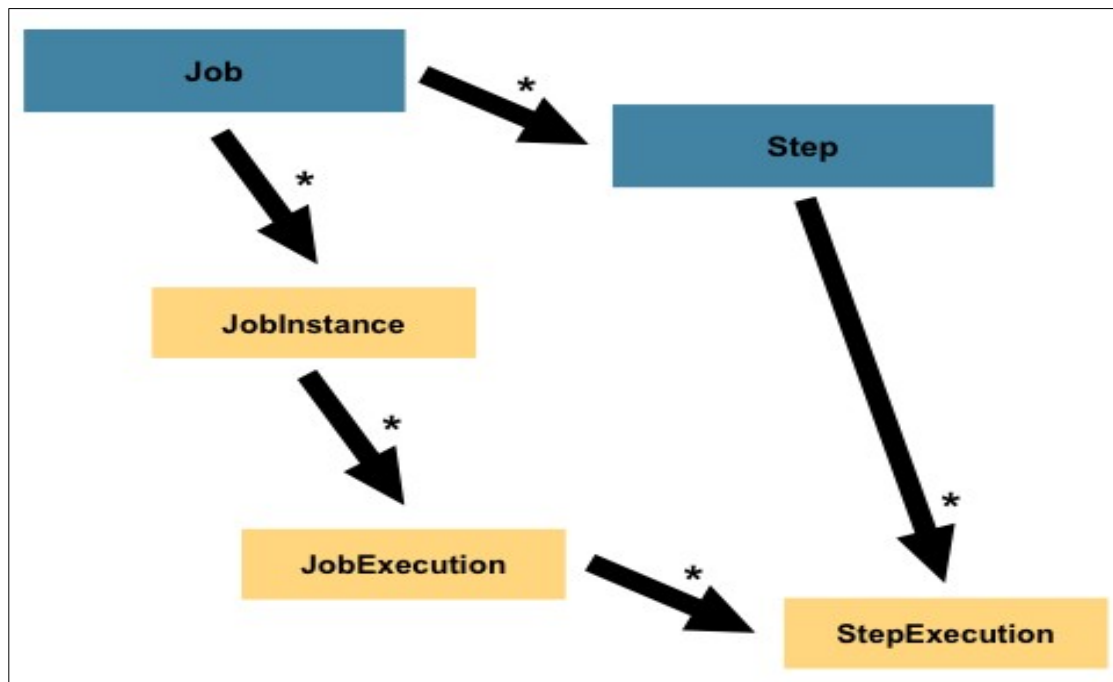


Job	Type de tâche à accomplir (paramétrage en Spring/Xml/Java)
JobInstance	Une instance planifiée de la tâche
JobParameter	Des paramétrages quelconques
JobExecution	Une exécution réelle d'une tâche (avec réussite ou échec)

Propriétés d'un "JobExecution":

status	A <code>BatchStatus</code> object that indicates the status of the execution. While running, it's <code>BatchStatus.STARTED</code> , if it fails, it's <code>BatchStatus.FAILED</code> , and if it finishes successfully, it's <code>BatchStatus.COMPLETED</code>
startTime	A <code>LocalDateTime</code> representing the current system time when the execution was started.
EndTime	A <code>LocalDateTime</code> representing the current system time when the execution finished, regardless of whether or not it was successful.
ExitStatus	The <code>ExitStatus</code> indicating the result of the run. It is most important because it contains an exit code that will be returned to the caller. See chapter 5 for more details.
CreateTime	A <code>LocalDateTime</code> representing the current system time when the <code>JobExecution</code> was first persisted. The job may not have been started yet (and thus has no start time), but it will always have a <code>createTime</code> , which is required by the framework for managing job level <code>ExecutionContexts</code> .
LastUpdated	A <code>LocalDateTime</code> representing the last time a <code>JobExecution</code> was persisted.
ExecutionContext	The 'property bag' containing any user data that needs to be persisted between executions.
FailureExceptions	The list of exceptions encountered during the execution of a <code>Job</code> . These

2.5. "Step" et "Step Execution"



Propriétés d'un "StepExecution":

status	A <code>BatchStatus</code> object that indicates the status of the execution. While it's running, the status is <code>BatchStatus.STARTED</code> , if it fails, the status is <code>BatchStatus.FAILED</code> , and if it finishes successfully, the status is <code>BatchStatus.COMPLETED</code>
startTime	A <code>LocalDateTime</code> representing the current system time when the execution was started.
EndTime	A <code>LocalDateTime</code> representing the current system time when the execution finished, regardless of whether or not it was successful.
ExitStatus	The <code>ExitStatus</code> indicating the result of the execution. It is most important because it contains an exit code that will be returned to the caller. See chapter 5 for more details.
ExecutionContext	The 'property bag' containing any user data that needs to be persisted between executions.
ReadCount	The number of items that have been successfully read
writeCount	The number of items that have been successfully written
commitCount	The number transactions that have been committed for this execution
rollbackCount	The number of times the business transaction controlled by the <code>Step</code> has been rolled back.
ReadSkipCount	The number of times <code>read</code> has failed, resulting in a skipped item.
ProcessSkipCount	The number of times <code>process</code> has failed, resulting in a skipped item.
FilterCount	The number of items that have been 'filtered' by the <code>ItemProcessor</code>
writeSkipCount	The number of times <code>write</code> has failed, resulting in a skipped item.

3. Evolution de Spring Batch (historique)

Version de SpringBatch (et époque)	Caractéristiques
V 1.x à partir de 2008	Configuration xml
V 2.x à partir de 2009 et 2010	Intégration de NoSql (mongoDB, ...) , configuration java (et pas seulement xml) , java ≥ 5
V 3.x à partir de Spring 4	Support de spring4 et java ≥ 8 , support du standard JSR-352 (lui même inspiré de SpringBatch) , SQLite support
V4.x à partir de Spring 5	Java 8 minimum , alignement avec spring 5
V5.x à partir de Spring 6	Java 17 minimum , alignement avec spring 6 et jarkata-ee

Autrement dit : **SpringBatch N** pour **Spring-Framework N-1** .

Old deprecated :

NB : depuis la version 4.3 de SpringBatch , ~~MapJobRepositoryFactoryBean~~ et beaucoup d'autres classes et interface en ~~MapXyz~~ sont devenues obsolètes et ont été carrément supprimées au sein de la version 5 de SpringBatch (allant de paire avec Spring framework 6).

Autrement dit , JobRepository (en version récente) nécessite absolument une persistance en base de données mais on peut éventuellement s'appuyer sur une persistance simple (in-memory embedded database like h2:mem).

4. Configuration maven sans springBoot

Pour Spring 5 et springBatch 4 :

```

<properties>
  <spring.version>5.3.31</spring.version>
  <spring.batch.version>4.3.10</spring.batch.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring.version}</version>
  </dependency> <!-- et indirectement spring-bean, spring-core , spring-aop -->

  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>${spring.version}</version>
  </dependency>

  ...
  <dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId> <!-- @PostConstruct -->
    <version>1.3.2</version>
  </dependency>

  <dependency>
    <groupId>org.springframework.batch</groupId>
    <artifactId>spring-batch-core</artifactId>
    <version>${spring.batch.version}</version>
  </dependency>

  <dependency>
    <groupId>org.springframework.batch</groupId>
    <artifactId>spring-batch-test</artifactId>
    <version>${spring.batch.version}</version><scope>test</scope>
  </dependency>
</dependencies>
...</project>

```

Pour Spring 6 et springBatch 5 :

```

<properties>
  <spring.version>6.1.2</spring.version>
  <spring.batch.version>5.1.1</spring.batch.version>
</properties>

... idem pour spring-batch-core , spring-batch-core , ....

  <dependency>
    <groupId>jakarta.annotation</groupId>
    <artifactId>jakarta.annotation-api</artifactId> <!-- @PostConstruct -->
    <version>2.1.1</version>
  </dependency>

```

5. Configuration maven moderne (avec springBoot)

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

<!-- h2 in memory (or ...) database is required for JobRepository persistance -->
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</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>
```

Compléments pour traiter des fichiers **xml** (avec Spring 6 , SpringBoot3 et SpringBatch 5) :

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-oxm</artifactId>
</dependency>
<!-- spring oxm is for object xml mapping -->

<dependency>
  <groupId>jakarta.xml.bind</groupId>
  <artifactId>jakarta.xml.bind-api</artifactId>
</dependency>

<dependency>
  <groupId>org.glassfish.jaxb</groupId>
  <artifactId>jaxb-runtime</artifactId>
</dependency>
```

Variation pour SpringBatch4 :

```
<dependency><groupId>javax.xml.bind</groupId>
  <artifactId>jaxb-api</artifactId></dependency>
```

6. Configurer et Lancer un "Job" simple

Deux types de configurations possibles (à choisir) :

- configuration explicite (sans springBoot) : `@EnableBatchProcessing`
- configuration implicite/automatique (avec SpringBoot) :

6.1. Rappel du contexte/cadre d'une application springBatch :

- Application en mode console (sans partie web)
- principal objectif : traitement de masse (batch)
- le dataSource principal et le transactionManager principal sont utilisés par les mécanismes internes de springBatch . Si besoin d'autres "datasource" , ces derniers seront secondaires.

6.2. Configuration explicite: "JobRepository" et "JobLauncher"

SpringBatch nécessite absolument une configuration d'une base de données pour ses mécanismes internes (persistance de JobRepository).

Voici un exemple de configuration explicite qui utilise volontairement des paramétrages standards (au plus près de springBoot)

src/main/resources/application.properties

```
#spring.datasource.url=jdbc:h2:mem:jobRepositoryDb
spring.datasource.url=jdbc:h2:~/jobRepositoryDb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
...
```

JobRepositoryBatchConfig.java

```
package tp.mySpringBatch.config;

import javax.sql.DataSource;
import org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import org.springframework.batch.core.repository.JobRepository;
import org.springframework.batch.core.repository.support.JobRepositoryFactoryBean;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.autoconfigure.batch.BatchDataSource;
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;
import org.springframework.core.io.ClassPathResource;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.jdbc.datasource.init.DataSourceInitializer;
import org.springframework.jdbc.datasource.init.ResourceDatabasePopulator;
import org.springframework.transaction.PlatformTransactionManager;

@Configuration
@EnableBatchProcessing()
public class JobRepositoryBatchConfig {
```

```

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

@Bean(name="dataSource") @Qualifier("batch")
@BatchDataSource
public DataSource batchDataSource(@Qualifier("batch") DataSourceProperties
                                batchDataSourceProperties) {
    return batchDataSourceProperties
        .initializeDataSourceBuilder()
        .build();
}

@Bean(name="transactionManager") @Qualifier("batch")
public PlatformTransactionManager batchTransactionManager(
    @Qualifier("batch") DataSource batchDataSource) {
    return new DataSourceTransactionManager(batchDataSource);
}

@Bean(name = "jobRepository")
public JobRepository jobRepository(
    @Qualifier("batch") DataSource batchDataSource,
    @Qualifier("batch") PlatformTransactionManager batchTransactionManager)
throws Exception {
    JobRepositoryFactoryBean factory = new JobRepositoryFactoryBean();
    factory.setDataSource(batchDataSource);
    factory.setTransactionManager(batchTransactionManager);
    factory.afterPropertiesSet();
    return factory.getObject();
}

/*
//THIS CONFIG IS THE SAME AS THE DEFAULT
// JobLauncher configuration (in spring-boot-batch) :*/
@Bean()
public JobLauncher jobLauncher(JobRepository jobRepository) throws Exception {
    TaskExecutorJobLauncher jobLauncher = new TaskExecutorJobLauncher(); //V5
    //SimpleJobLauncher jobLauncher = new SimpleJobLauncher(); //V4
    jobLauncher.setJobRepository(jobRepository);
    jobLauncher.afterPropertiesSet();
    return jobLauncher;
}
}

```

NB. (possible MAIS pas conseillé) :

Seulement en V5 de SpringBatch , il est possible de configurer se configurer un "SpringBatch maison" (`spring.batch.datasource.url` plutôt que `spring.datasource.url` , `batchDataSource` plutôt que `dataSource` , `batchTransactionManager` plutôt que `transactionManager` , `@EnableBatchProcessing` avec `dataSourceRef` et `transactionManagerRef`) de manière à éviter tout potentiel conflit avec d'autres parties de l'application.

```

@Configuration
@EnableBatchProcessing(dataSourceRef = "batchDataSource",
    transactionManagerRef = "batchTransactionManager")
public class JobRepositoryBatchConfig {
    /*
        NB: in application.properties
        NOT spring.datasource.url=jdbc:h2:mem:jobRepositoryDb
        BUT spring.batch.datasource.url=jdbc:h2:~/jobRepositoryDb
        and spring.batch.datasource.username=sa
        spring.batch.datasource.password=
    */
    @Bean @Qualifier("batch")
    @ConfigurationProperties("spring.batch.datasource")
    public DataSourceProperties batchDataSourceProperties() {
        return new DataSourceProperties();
    }

    @Bean(name="batchDataSource") @Qualifier("batch")
    @BatchDataSource
    public DataSource batchDataSource(@Qualifier("batch") DataSourceProperties
        batchDataSourceProperties) {
        return batchDataSourceProperties
            .initializeDataSourceBuilder()
            .build();
    }

    @Bean(name="batchTransactionManager") @Qualifier("batch")
    public PlatformTransactionManager batchTransactionManager(
        @Qualifier("batch") DataSource batchDataSource) {
        return new DataSourceTransactionManager(batchDataSource);
    }
}

```

La V4 de `@EnableBatchProcessing` ne comporte pas les paramètres `dataSourceRef` et `transactionManagerRef` !!!!

Variante sans springBoot

```

@Configuration
@PropertySource("classpath:application.properties")
@EnableBatchProcessing()
public class JobRepositoryBatchConfig {

    @Value("${spring.datasource.driverClassName}")
    private String jdbcDriver;

    @Value("${spring.datasource.url}")
    private String dbUrl;

    @Value("${spring.datasource.username}")
    private String dbUsername;

    @Value("${spring.datasource.password}")
    private String dbPassword;

    @Bean(value = "dataSource") @Qualifier("jobRepositoryDb")
    @Primary
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(jdbcDriver);
        dataSource.setUrl(dbUrl);
        dataSource.setUsername(dbUsername);
        dataSource.setPassword(dbPassword);
        return dataSource;
    }

    ...}

```

NB :

Possible mais intéressant que sans springBoot :

@Bean @Qualifier("batch")

```
public DataSourceInitializer databasePopulator(
    @Qualifier("batch") DataSource batchDataSource) {
    ResourceDatabasePopulator populator = new ResourceDatabasePopulator();
    //populator.addScript(new ClassPathResource("org/springframework/batch/core/schema-drop-h2.sql"));
    populator.addScript(new ClassPathResource("org/springframework/batch/core/schema-h2.sql"));
    populator.setContinueOnError(false); populator.setIgnoreFailedDrops(false);
    DataSourceInitializer dataSourceInitializer = new DataSourceInitializer();
    dataSourceInitializer.setDataSource(batchDataSource);
    dataSourceInitializer.setDatabasePopulator(populator);
    return dataSourceInitializer;
}
```

Un éventuel bean de type **DataSourceInitializer** peut servir à déclencher un script SQL (**org/springframework/batch/core/schema-h2.sql**) permettant de préparer/créer des tables nécessaires au fonctionnement de SpringBatch .

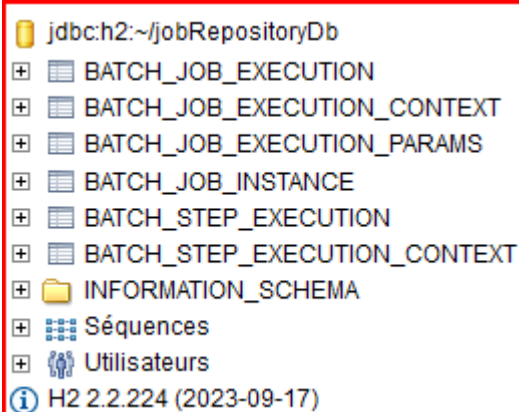
NB : Avec springBoot, ceci peut quelquefois être fait (en mieux : qu'au premier démarrage) , en plaçant l'option

#spring.batch.initialize-schema=always pour SpringBoot 2.0

#spring.batch.jdbc.initialize-schema=always pour SpringBoot >=2.5 (ex : 3)

spring.batch.jdbc.initialize-schema=always

dans *application.properties*.



Exécuter

Run Selected

Complètement automatique

Effacer

Instruction SQL:

SELECT * FROM BATCH_JOB_EXECUTION

SELECT * FROM BATCH_JOB_EXECUTION;

JOB_EXECUTION_ID	VERSION	JOB_INSTANCE_ID	CREATE_TIME	START_TIME	END_TIME	STATUS	EXIT_CODE
1	2	1	2024-02-20 10:04:59.500495	2024-02-20 10:04:59.5364745	2024-02-20 10:04:59.7065685	COMPLETED	COMPLETED

6.3. Configuration implicite de springBatch via springBoot

En exploitant un peu plus la configuration automatique de **spring-boot-starter-batch** on peut considérer qu'une grande partie de la configuration de SpringBatch sera automatiquement disponible selon les règles suivantes :

- un DataSource et TransactionManager seront initialisés/configurés par défaut via spring-boot-starter-batch
- un JobRepository sera automatiquement construit par spring-boot-starter-batch au dessus des dataSource et transactionManager par défaut
- un JobLauncher sera automatiquement construit par spring-boot-starter-batch au dessus de JobRepository
- DE façon A POUVOIR CONSTRUIRE TOUT ça , SpringBoot (en tant que config par défaut) doit construire lui même ses "dataSource" et "transactionManager" par défaut
 - > pas d'autres "DataSource" doivent être détectées au démarrage d'une SpringBatchApp basée sur spring-boot-starter-batch !!!
 - > Ou bien si plusieurs "Datasources" doivent coexister SEULEMENT UNE DataSource (avec @Primary et @BatchDataSource) doit être explicitement configurée (avec même comportement que par défaut avec springBatch) de manière à permettre à spring-boot-starter-batch de construire sa pile de composants

AutomaticSpringBootTestJobRepositoryConfig.java

```
...
@Configuration
//SIMPLE éventuel complément ou redéfinition partielle de la
//configuration automatique apportée par spring-boot-starter-batch
public class AutomaticSpringBootTestJobRepositoryConfig {

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

    //NB: cette configuration explicite du "datasource" principal (@Primary)
    // à utiliser par springBatch et springBoot (@BatchDataSource) n'est utile
    // que pour préciser la configuration "DataSource" prioritaire (sans conflit avec les autres)
    @Primary
    @BatchDataSource
    @Bean(value = "dataSource")
    public DataSource dataSource(@Qualifier("jobRepositoryDb")
                                DataSourceProperties dbDataSourceProperties) {
        return dbDataSourceProperties.initializeDataSourceBuilder().build();
    }
}
```

NB : l'option ***spring.batch.initialize-schema=always*** permet de créer automatiquement les tables nécessaires au fonctionnement de SpringBatch au sein de la base de données associée au dataSource marqué par @Primary et @BatchDataSource .

application.properties

```
#main jobRepository DataBase for SpringBatch:
spring.datasource.url=jdbc:h2:~/jobRepositoryDb
spring.datasource.username=sa
spring.datasource.password=

#secondary DataBases for some Jobs:
spring.otherdb.datasource.url=jdbc:h2:~/otherDb
spring.otherdb.datasource.username=sa
spring.otherdb.datasource.password=

#disable auto launching of jobs (spring-boot-starter-batch)
spring.batch.job.enabled=false

#automatic create_table for jobRepository (spring-boot-starter-batch):
#spring.batch.jdbc.initialize-schema=never
#spring.batch.jdbc.initialize-schema=embedded
spring.batch.jdbc.initialize-schema=always
```

Variantes de configurations attendues selon les versions :

Avec SpringBatch 4 et SpringBoot 2 : @SpringBootApplication() et @EnableBatchProcessing()
Avec SpringBatch 5 et SpringBoot 3 : @SpringBootApplication() sans @EnableBatchProcessing()

```
#spring.batch.initialize-schema=always pour SpringBoot 2.0
#spring.batch.jdbc.initialize-schema=always pour SpringBoot >=2.5 (ex : 3)
```

6.4. Configuration minimaliste d'un tasklet et d'un job en java

tp.tpSpringBatch.tasklet.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;}
}
```

tp.tpSpringBatch.tasklet.bean.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");
    }
}
```

tp.tpSpringBatch.job.java.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 /* @Qualifier("batch") */
    protected PlatformTransactionManager batchTxManager;

    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.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 {

    @Bean(name="myHelloWorldJob")
    public Job myHelloWorldJob(
        @Qualifier("simplePrintMessageStep") 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();
    }
}
```

Variations syntaxiques pour SpringBatch 4 :

```
var jobBuilder = (new JobBuilder(name)).repository(jobRepository);
```

```
var stepBuilder = (new StepBuilder(name)).repository(jobRepository);
return stepBuilder
    .transactionManager(batchTxManager)
    ...
    .build() ;
```

6.5. Lancement d'un job depuis java en ligne de commande

```
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

//avec en plus @EnableBatchProcessing() si ancienne version

```
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 ...");
}
}
```

Variante sans springBoot :

```
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.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.stereotype.Component;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = { "tp.tpSpringBatch" })
public class MyMainSpringBatchConfig {
}

//version sans springBoot
@Component
public class TpSpringBatchApplicationWithoutSpringBoot {

    private final JobLauncher jobLauncher;
    private final ApplicationContext applicationContext;

    @Autowired
    public TpSpringBatchApplicationWithoutSpringBoot(JobLauncher jobLauncher,
                                                    ApplicationContext applicationContext) {
        this.jobLauncher = jobLauncher;
        this.applicationContext = applicationContext;
    }

    /*
    NB: sans automatisme springBoot/spring.batch.initialize-schema=always
    il faut SEULEMENT au PREMIER LANCEMENT décommenter //@Bean @Qualifier("batch")
    au dessus de public DataSourceInitializer databasePopulator ...
    dans tp.tpSpringBatch.config.JobRepositoryBatchConfig
    */

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

```
ApplicationContext /*AnnotationConfigApplicationContext*/ springContext = new
    AnnotationConfigApplicationContext(MyMainSpringBatchConfig.class) ;

TpSpringBatchApplicationWithoutSpringBoot app =
    springContext.getBean(TpSpringBatchApplicationWithoutSpringBoot.class);

app.run(args);
}

public void run(String... args) throws Exception {
    //même code que la principale variante (avec springBoot)
}
}
```

6.6. Configuration d'un job en xml

En 2024 ou plus , la configuration principale d'une application spring est souvent en java. On peut néanmoins envisager une sous configuration partielle en XML (sur les parties "Step/Job" et "Reader/Writer") .

tp.tpSpringBatch.job.xml.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 {
}
```

src/main/resources/job/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">

    <!-- <alias name="batchTransactionManager" alias="transactionManager" /> -->
    <!-- listeners -->
    ...
    <!-- readers and writers -->
</beans>
```

src/main/resources/job/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 ...
```

Plus trop à la mode mais encore possible (selon configuration xml) :

Lancement (à l'ancienne) d'un batch depuis une ligne de commande et depuis un fichier de configuration principal en xml :

```
java ... org.springframework.batch.core.launch.support.CommandLineJobRunner
springFileConfOfJob.xml
JobName
```

indirectement via maven:

mvn exec:java -

Dexec.mainClass=org.springframework.batch.core.launch.support.**CommandLineJobRun**
ner

-*Dexec.args*="*simpleJob.xml helloWorldJob*"

6.7. Lancement d'un job depuis un test JUnit

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.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.test.context.ActiveProfiles;
import tp.tpSpringBatch.TpSpringBatchApplication;
import tp.tpSpringBatch.tasklet.bean.PrintHelloWorldMessageTaskletBean;
```

@Configuration

//@EnableBatchProcessing //nécessaire avec SpringBoot2/SpringBatch4

@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 pratice in V5 , mandatory in SprinBatch 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());
}
```

Variante de test unitaire pour job configuré en XML :

```
@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 {
...
}
```

Variante de test unitaire sans springBoot :

```
@Configuration
@Import({JobRepositoryBatchConfig.class,
        HelloWorldJobConfig.class ,
        PrintHelloWorldMessageTaskletBean.class})
class HelloWorldJobTestConfig{
}

@SpringBatchTest
@SpringJUnitConfig({ HelloWorldJobTestConfig.class } )
@ActiveProfiles(profiles = {})
public class TestHelloWorldJob {
...
}
```

II - Tasklet et Chunk (CSV, XML, JSON, ...)

1. Tasklets et chunks

Un "Step" (étape d'un job) peut soit être implémenté comme un simple **tasklet** (assez élémentaire) , soit être implémenté via la mise en œuvre d'un "**chunk**" (cas évolué le plus courant).

1.1. Exemples d'utilisation de tasklets :

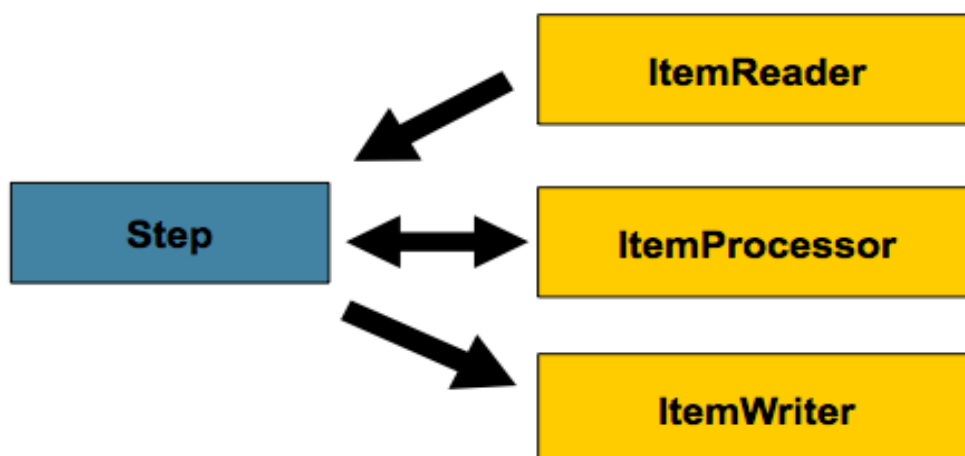
- Exemples élémentaires ("hello world" et variantes)
- Appels d'une procédure stockées traitant elle même un grand ensemble de données
- Appel d'un web service
- Unzip d'un fichier
- Tout autre traitement élémentaire (qui ne se décompose pas)

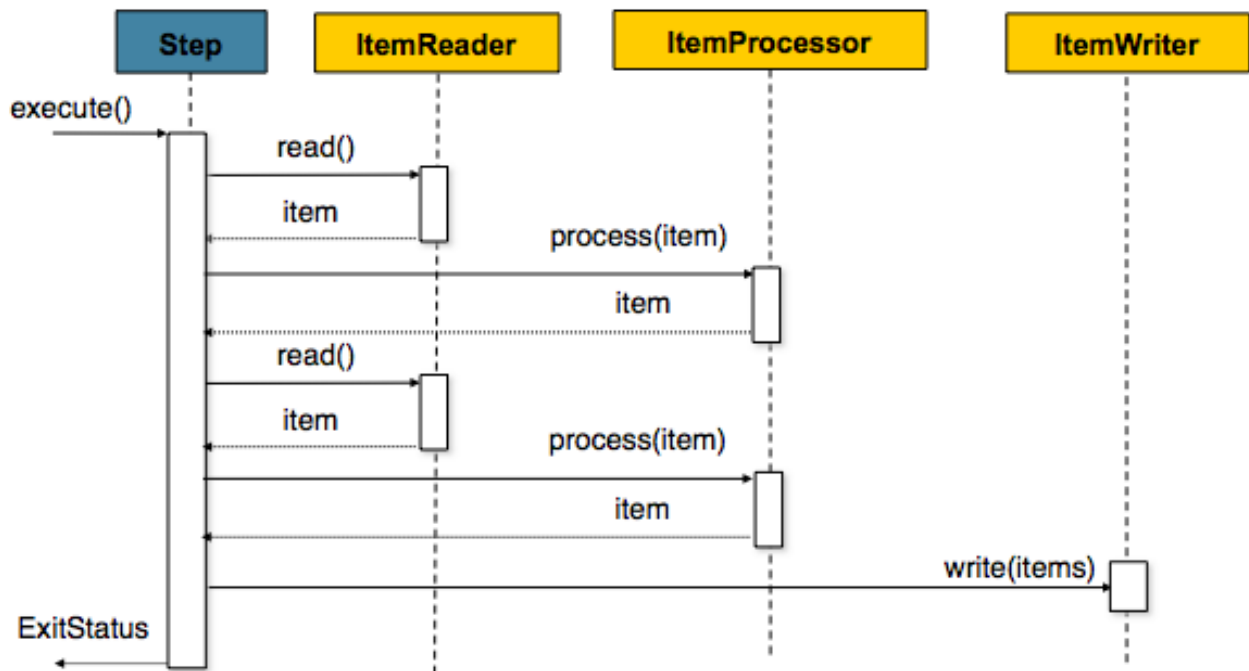
1.2. Exemples d'utilisation des chunks

- Lectures/écritures en boucles avec ou sans traitements intermédiaires
- Toute autre opération évoluée

2. Orientation "chunk"

2.1. Traitement par blocs ("chunks")





Lectures et traitements unitaires puis écriture par lots . Le "**commitInterval**" défini la taille du lot.
Gestion de transaction (commit/rollback)

algorithme prédéfini :

```

List items = new ArrayList();
for(int i = 0; i < commitInterval; i++){
    Object item = itemReader.read(); //lecture unitaire
    Object processedItem = itemProcessor.process(item); //traitement(s)
    items.add(processedItem);
}
itemWriter.write(items); // écriture par paquet de "commit-interval"

```

2.2. "ItemProcessor" simple

Implémentations élémentaires:

MyProductProcessor.java

```

//ItemProcessor<I,O> I=InputItemType , O=OutputItemType
public class SimpleUppercasePersonProcessor implements ItemProcessor<Person,Person>{

    public Person process(Person pers) throws Exception {
        Person person=new Person(pers.getFirstName(),
                                pers.getLastName().toUpperCase() ,
                                pers.getAge(),pers.getActive());

        return person;
    }
}

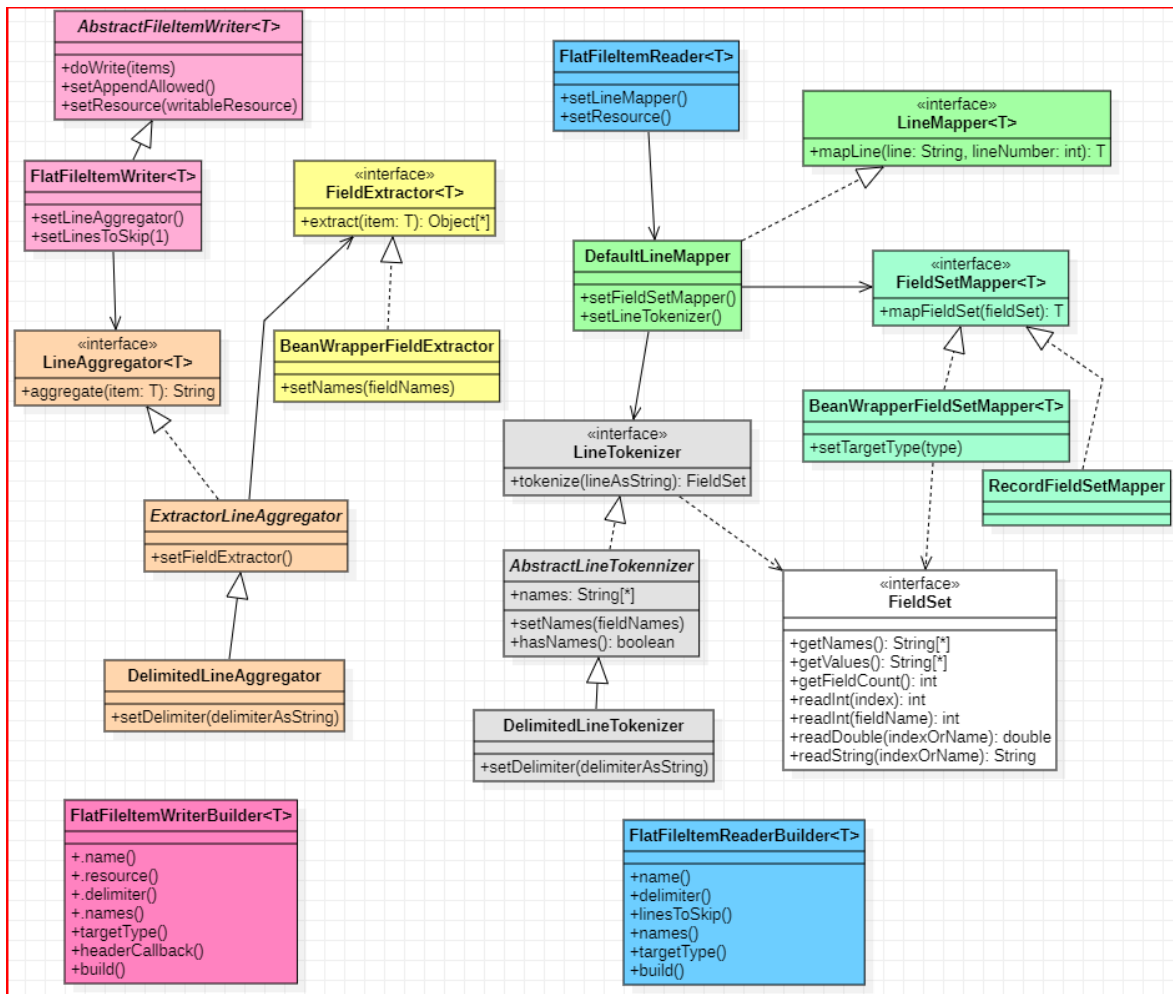
```

3. Lecture/écriture de fichiers avec spring-batch

3.1. Gestion des fichiers plats (.txt , .csv , ...)

"Spring Batch" utilise une combinaison des éléments suivants pour traiter les fichiers plats:

FlatFileItemReader	ItemReader pour les fichiers plats (avec lignes à lire séquentiellement et avec champs délimités par séparateur ou bien de longueurs fixes) .
FieldSet	équivalent d'un "RecordSet" mais pour les fichiers ; Paquet de champs à lire sur une ligne d'un fichier plat. Structuré avec champs typés , numérotés/indexés et/ou nommés .
LineMapper	Map entre numéroDeLigne et Ligne à lire/analyser
LineTokenizer	Objet technique pour découper une ligne en champ (selon délimiteur ou positions fixes)
FieldSetMapper	Objet technique associant un "FieldSet" à une structure d'objet "java"
FlatFileItemWriter	ItemWriter pour les fichiers plats (avec lignes à lire séquentiellement et avec champs délimités par séparateur ou bien de longueurs fixes) .
LineAggregator	Objet technique ajoutant sur une ligne d'un fichier à écrire les différentes parties (champs) à générer et à séparer via un délimiteur , soit à placer à des positions fixes.
FieldExtractor	Objet technique qui extraie certains champs (à écrire) d'un objet java



3.2. Gestion des .csv (avec enregistrements délimités (";"))

Exemple de fichier csv :

data/input/csv/inputData.csv

```

firstName;lastName;age;active
Jean;Bon;41;true
Alex;Therieur;42;false
Axelle;Aire;33;true
Olie;Condor;44;false
    
```

Reader "csv" configuré en xml :

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:util="http://www.springframework.org/schema/util"
  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
    http://www.springframework.org/schema/util
    https://www.springframework.org/schema/util/spring-util.xsd">
    
```

```
<bean id = "personCsvFileReader"
  class = "org.springframework.batch.item.file.FlatFileItemReader">
  <property name = "resource" value = "file:data/input/csv/inputData.csv" />
  <property name = "linesToSkip" value = "1" />
  <property name = "lineMapper">
    <bean
      class = "org.springframework.batch.item.file.mapping.DefaultLineMapper">
      <property name = "lineTokenizer">
        <bean
          class = "org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
          <property name = "names" value="firstName,lastName,age,active">
          </property>
          <property name = "delimiter" value = ";" />
          </bean>
        </property>
      <property name = "fieldSetMapper">
        <bean class =
          "org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper" >
          <property name = "targetType" value = "tp.mySpringBatch.model.Person" />
          </bean>
        </property>
      </bean>
    </property>
  </bean>
</property>
</bean>
</beans>
```

Reader "csv" configuré en java :

```
...
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
...

@Configuration
@Profile("!xmlJobConfig")
public class MyCsvFilePersonReaderConfig {

    @Value("file:data/input/csv/inputData.csv") //to read in project root directory
    private Resource inputCsvResource;

    //V2 with FlatFileItemReaderBuilder
    @Bean @Qualifier("csv")
    @JobScope
    public FlatFileItemReader<Person> personCsvFileReader() {

        return new FlatFileItemReaderBuilder<Person>()
            .name("personCsvFileReader")
            .resource(inputCsvResource)
```



```

        .linesToSkip(1)
        .delimited()
        .delimiter(";")
        .names("firstName", "lastName", "age", "active")
        .targetType(Person.class)
        .build();
    }

    /*
    //OLD V1 with sub-methods and without builder :

    @Bean @Qualifier("csv")
    public FlatFileItemReader<Person> personCsvFileReader() {
        var personCsvFileItemReader = new FlatFileItemReader<Person>();

        personCsvFileItemReader.setLineMapper(
            this.personLineMapper(this.personLineTokenizer(),
                this.personFieldSetMapper()));

        personCsvFileItemReader.setResource(inputCsvResource);

        personCsvFileItemReader.setLinesToSkip(1);

        return personCsvFileItemReader;
    }

    public DefaultLineMapper<Person> personLineMapper(
        LineTokenizer personLineTokenizer,
        FieldSetMapper<Person> personFieldSetMapper) {
        var lineMapper = new DefaultLineMapper<Person>();
        lineMapper.setLineTokenizer(personLineTokenizer);
        lineMapper.setFieldSetMapper(personFieldSetMapper);
        return lineMapper;
    }

    public BeanWrapperFieldSetMapper<Person> personFieldSetMapper() {
        var fieldSetMapper = new BeanWrapperFieldSetMapper<Person>();
        fieldSetMapper.setTargetType(Person.class);
        return fieldSetMapper;
    }

    public DelimitedLineTokenizer personLineTokenizer() {
        var tokenizer = new DelimitedLineTokenizer();
        tokenizer.setDelimiter(";");
        tokenizer.setNames("firstName", "lastName", "age", "active");
        return tokenizer;
    }
    */
}

```

Writer "Csv" configuré en xml :

```
...
<bean id="csvFilePersonWriter"
class="org.springframework.batch.item.file.FlatFileItemWriter">
  <property name="resource" value="file:data/output/csv/outputData.csv" />
  <property name="appendAllowed" value="false" />
  <property name="lineAggregator">
    <bean class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
      <property name="delimiter" value=";" />
      <property name="fieldExtractor">
        <bean
          class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor">
            <property name="names" value="firstName,lastName,age,active" />
          </bean>
        </property>
      </bean>
    </property>
  </bean>
</property>
</bean>
</property>
</bean>
```

Writer "Csv" configuré en java :

```
package tp.mySpringBatch.writer.java;
import org.springframework.batch.item.file.FlatFileItemWriter;
import org.springframework.batch.item.file.builder.FlatFileItemWriterBuilder;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.core.io.WritableResource;
import tp.mySpringBatch.model.Person;

@Configuration
@Profile("!xmlJobConfig")
public class MyCsvFilePersonWriterConfig {

    @Value("file:data/output/csv/outputData.csv") //to read in project root directory
    private WritableResource outputCsvResource;

    //V2 with builder:
    @Bean @Qualifier("csv")
    FlatFileItemWriter<Person> csvFilePersonWriter() {

        return new FlatFileItemWriterBuilder<Person>()
            .name("csvFilePersonWriter")
            .resource(outputCsvResource)
            .delimited()
            .delimiter(";")
            .names("firstName", "lastName", "age", "active")
            .headerCallback((writer)-> {writer.write("firstname;lastname;age;active");})
            .build();
    }
}
```

```

}

/*
//VI without builder and with sub methods:

@Bean @Qualifier("csv")
FlatFileItemWriter<Person> personCsvFileItemWriter() {

    //Create writer instance
    FlatFileItemWriter<Person> writer = new FlatFileItemWriter<>();

    //Set output file location
    //WritableResource outputCsvResource = new
        FileSystemResource("data/output/csv/outputData.csv");
    writer.setResource(outputCsvResource);

    //All job repetitions should "append" to same output file
    //writer.setAppendAllowed(true);
    writer.setAppendAllowed(false);

    writer.setLineAggregator(this.personLineAggregator(this.personFieldExtractor()));
    return writer;
}

FieldExtractor<Person> personFieldExtractor(){
    BeanWrapperFieldExtractor<Person> beanWrapperFieldExtractor =
        new BeanWrapperFieldExtractor<>();
    beanWrapperFieldExtractor.setNames(new String[]{"firstName", "lastName",
        "age", "active"});

    return beanWrapperFieldExtractor;
}

LineAggregator<Person> personLineAggregator(
    FieldExtractor<Person> personFieldExtractor){
    DelimitedLineAggregator<Person> delimitedLineAggregator =
        new DelimitedLineAggregator<>();
    delimitedLineAggregator.setDelimiter(";");
    delimitedLineAggregator.setFieldExtractor(personFieldExtractor);
    return delimitedLineAggregator;
}
*/
}

```

Exemples de Jobs configurés en XML:

commonConfig.xml

```

...
<import resource="classpath:job/rw/csvReader.xml" />
<import resource="classpath:job/rw/csvWriter.xml" />
...

```

fromXmlToCsvJob.xml

```
...
<job id="fromXmlToCsvJob" xmlns="http://www.springframework.org/schema/batch">
    <step id="step1_of_fromXmlToCsvJob" >
        <tasklet>
            <chunk reader="personXmlFileItemReader" writer="csvFilePersonWriter"
                processor="simpleUppercasePersonProcessor" commit-interval="1" />
        </tasklet>
    </step>
</job>
```

fromCsvToJsonJob.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:util="http://www.springframework.org/schema/util"
    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
        http://www.springframework.org/schema/util
        https://www.springframework.org/schema/util/spring-util.xsd">

    <job id="fromCsvToJsonJob" xmlns="http://www.springframework.org/schema/batch">
        <step id="step1_of_fromCsvToJsonJob" >
            <tasklet>
                <chunk reader="personCsvFileReader" writer="personJsonFileItemWriter"
                    processor="simpleUppercasePersonProcessor" commit-interval="1" />
            </tasklet>
        </step>
    </job>
</beans>
```

Exemples de Jobs configurés en Java:

MyBasicCsvToXmlJobConfig.java

```
....

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

    @Bean
    public Job fromCsvToXmlJob(@Qualifier("csvToXml") Step step1) {
        var name = "Persons CsvToXml Job";
        var jobBuilder = new JobBuilder(name, jobRepository);
        return jobBuilder.start(step1)
            .listener(new JobCompletionNotificationListener())
            .build();
    }

    @Bean @Qualifier("csvToXml")
```

```
public Step stepCsvToXml(@Qualifier("csv") ItemReader<Person> personItemReader,
                        @Qualifier("xml") ItemWriter<Person> personItemWriter,
                        SimpleUppercasePersonProcessor simpleUppercasePersonProcessor ) {
    var name = "COPY CSV RECORDS To another CSV Step";
    var stepBuilder = new StepBuilder(name, jobRepository);
    return stepBuilder
        .<Person, Person>chunk(5, batchTxManager)
        .reader(personItemReader)
        .processor(simpleUppercasePersonProcessor)
        .writer(personItemWriter)
        .build();
}
```

MyBasicXmlToCsvJobConfig.java

```
....
@Qualifier("csv") ItemWriter<Person> personItemWriter
```

Variations syntaxiques pour SpringBatch 4 :

```
var stepBuilder = (new StepBuilder(name)).repository(jobRepository);
return stepBuilder
    .transactionManager(batchTxManager)
    .<Person, Person>chunk(5)....build() ;
```

3.3. Gestion des fichiers plats à positions fixes

Exemple de fichier .txt à positions fixes:

data/input/txt/fixedPositionInputData.txt

Jean	Bon	41	true
Alain	Therieur	42	false
Axelle	Aire	33	true
Olie	Condor	44	false

Reader "fixedPosTxt" configuré en xml :

```
...
<bean id="personFixedPosTxtFileReader"
class="org.springframework.batch.item.file.FlatFileItemReader">
  <property name="resource" value="file:data/input/txt/fixedPositionInputData.txt" />
  <property name="lineMapper">
    <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
      <property name="fieldSetMapper">
        <bean
          class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper">
            <property name="targetType" value="tp.mySpringBatch.model.Person" />
          </bean>
        </property>
      <property name="lineTokenizer">
        <bean class="org.springframework.batch.item.file.transform.FixedLengthTokenizer">
          <property name="names" value="firstName,lastName,age,active" />
          <property name="columns" value="1-12, 25-48, 49-52, 53-58" />
        </bean>
      </property>
    </bean>
  </property>
</bean>
```

Reader "fixedPosTxt" configuré en java :

```
package tp.mySpringBatch.reader.java;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.batch.item.file.transform.Range;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.core.io.Resource;
import tp.mySpringBatch.model.Person;

@Configuration
@Profile("!xmlJobConfig")
public class MyFixedPosTextFilePersonReaderConfig {

    @Value("file:data/input/txt/fixedPositionInputData.txt") //to read in project root directory
    private Resource inputTxtResource;
```

```
//V2 with FlatFileItemReaderBuilder
@Bean @Qualifier("fixedPosTxt")
public FlatFileItemReader<Person> personFixedPosTxtFileReader() {

    return new FlatFileItemReaderBuilder<Person>()
        .name("personFixedPosTxtFileReader")
        .resource(inputTxtResource)
        .targetType(Person.class)
        .fixedLength()
        .columns(new Range(1, 24), new Range(25, 48), new Range(49, 52), new Range(53, 58))
        .names("firstName", "lastName", "age", "active")
        .build();
}

/*
//OLD V1 with sub-methods and without builder :
@Bean @Qualifier("fixedPosTxt")
public FlatFileItemReader<Person> personFixedPosTxtFileReader() {
    var personFixPosTxtFileItemReader = new FlatFileItemReader<Person>();

    personFixPosTxtFileItemReader.setLineMapper(
        this.personLineMapper(this.personLineFixedLengthTokenizer(),
            this.personFieldSetMapper()));

    personFixPosTxtFileItemReader.setResource(inputTxtResource);

    return personFixPosTxtFileItemReader;
}

public DefaultLineMapper<Person> personLineMapper(LineTokenizer personLineTokenizer,
    FieldSetMapper<Person> personFieldSetMapper) {
    var lineMapper = new DefaultLineMapper<Person>();
    lineMapper.setLineTokenizer(personLineTokenizer);
    lineMapper.setFieldSetMapper(personFieldSetMapper);
    return lineMapper;
}

public BeanWrapperFieldSetMapper<Person> personFieldSetMapper() {
    var fieldSetMapper = new BeanWrapperFieldSetMapper<Person>();
    fieldSetMapper.setTargetType(Person.class);
    return fieldSetMapper;
}

public FixedLengthTokenizer personLineFixedLengthTokenizer() {
    var tokenizer = new FixedLengthTokenizer();

    tokenizer.setNames("firstName", "lastName", "age", "active");
    tokenizer.setColumns(new Range(1, 24),
        new Range(25, 48),
```

```

        new Range(49, 52),
        new Range(53, 58));

    return tokenizer;
}
*/
}

```

Writer "fixedPosTxt" configuré en xml :

```

<bean id="fixedPosTxtFilePersonWriter"
class="org.springframework.batch.item.file.FlatFileItemWriter">
  <property name="resource" value="file:data/output/txt/fixedPositionOutputData.txt" />
  <property name="lineAggregator">
    <bean class="org.springframework.batch.item.file.transform.FormatterLineAggregator">
      <property name="minimumLength" value="58" />
      <property name="maximumLength" value="58" />
      <property name="format" value="%-24s%-24s%-4d%-6b" />
      <property name="fieldExtractor">
        <bean
          class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor">
            <property name="names" value="firstName,lastName,age,active" />
          </bean>
        </property>
      </bean>
    </property>
  </bean>
</property>
</bean>

```

Writer "fixedPosTxt" configuré en java :

```

package tp.mySpringBatch.writer.java;
import org.springframework.batch.item.file.FlatFileItemWriter;
import org.springframework.batch.item.file.builder.FlatFileItemWriterBuilder;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.core.io.WritableResource;
import tp.mySpringBatch.model.Person;

@Configuration
@Profile("!xmlJobConfig")
public class MyFixedPosTxtFilePersonWriterConfig {

    @Value("file:data/output/txt/fixedPositionOutputData.txt") //to read in project root directory
    private WritableResource outputTxtResource;

    // V2 via builder:
    @Bean @Qualifier("fixedPosTxt")
    FlatFileItemWriter<Person> fixedPosTxtFilePersonWriter() {
        return new FlatFileItemWriterBuilder<Person>()
            .name("fixedPosTxtFilePersonWriter")

```



```

        .resource(outputTxtResource)
        .formatted()
        .maxLength(58)
        .minLength(58)
        .format("%-24s%-24s%-4d%-6b")
        .names("firstName", "lastName", "age", "active")
        .build();
    }

    /*
    // V1 without builder:
    @Bean @Qualifier("fixedPosTxt")
    FlatFileItemWriter<Person> fixedPosTxtFilePersonWriter() {

        //Create writer instance
        FlatFileItemWriter<Person> writer = new FlatFileItemWriter<>();

        //Set output file location
        writer.setResource(outputTxtResource);

        //All job repetitions should "append" to same output file
        //writer.setAppendAllowed(true);
        writer.setAppendAllowed(false);

        writer.setLineAggregator(this.personLineAggregator(this.personFieldExtractor()));
        return writer;
    }

    FieldExtractor<Person> personFieldExtractor(){
        BeanWrapperFieldExtractor<Person> beanWrapperFieldExtractor =
            new BeanWrapperFieldExtractor<>();
        beanWrapperFieldExtractor.setNames(new String[]{"firstName", "lastName",
            "age", "active"});
        return beanWrapperFieldExtractor;
    }

    FormatterLineAggregator<Person> personLineAggregator(
        FieldExtractor<Person> personFieldExtractor){

        FormatterLineAggregator<Person> formatterLineAggregator =
            new FormatterLineAggregator<>();
        formatterLineAggregator.setMinLength(58);
        formatterLineAggregator.setMaxLength(58);
        formatterLineAggregator.setFormat("%-24s%-24s%-4d%-6b");
        formatterLineAggregator.setFieldExtractor(personFieldExtractor);
        return formatterLineAggregator;
    }
    */
}

```

Exemples de Jobs configurés en XML:

commonConfig.xml

```
...
<import resource="classpath:job/rw/fixedPosTxtReader.xml" />
<import resource="classpath:job/rw/fixedPosTxtWriter.xml" />
...
```

fromFixedPosToCsvJob.xml

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

    <step id="step1_of_fromFixedPosTxtToCsvJob" >
        <tasklet>
            <chunk reader="personFixedPosTxtFileReader" writer="csvFilePersonWriter"
                processor="simpleUppercasePersonProcessor" commit-interval="1" />
        </tasklet>
    </step>
...
```

fromCsvToFixedPosTxtJob.xml

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

    <step id="step1_of_fromCsvToFixedPosTxtJob" >
        <tasklet>
            <chunk reader="personCsvFileReader" writer="fixedPosTxtFilePersonWriter"
                processor="simpleUppercasePersonProcessor" commit-interval="1" />
        </tasklet>
    </step>
...
```

A utiliser dans des jobs configurés en Java:

```
@Qualifier("fixedPosTxt") ItemWriter<Person> personItemWriter
```

et

```
@Qualifier("fixedPosTxt") ItemReader<Person> personItemReader
```

3.4. Gestions de fichiers ".csv / .txt" avec format Hybride

Cas d'un fichier avec lignes différentes (avec "switch" selon début)

lineMapper = new **PatternMatchingCompositeLineMapper**();
avec exemple partiel au sein de la documentation de référence de "SpringBatch" .

Cas d'un fichier où certaines sous parties sont elles mêmes à décomposer

Certains fichiers ".csv" peuvent éventuellement comporter des **sous parties** qui sont :

- soit à décomposer avec un "sous-délimiteur"
- soit à interpréter en mode "positions fixes"

Exemple :

personWithNumAndAddress.csv

```
id;firstName;lastName;age;active;socialSecurityNumber;address
1;Jean;Bon;41;true;101020300400518;FRA!75001!Paris!rue_xy!2!1er etage
2;Axelle;Aire;33;true;201020300400565;FRA!76000!Rouen!rue xy!4!1er etage
```

avec le numéro de sécurité social qui se décompose lui même selon des positions fixes :

```
101020300400518
```

et l'adresse qui se décompose en analysant se sous délimiteur " !" :

```
FRA!75001!Paris!rue_xy!2!1er etage
```

Structure java :

```
public class PersonWithNumAndAddress extends Person {
    private SocialSecurityNumber socialSecurityNumber;
    private Address address;
    ...
}
```

```
public class SocialSecurityNumber {
    private String genre ; // sur 1 caractère ("1" pour Homme et "2" pour Femme )
    private String anneeNaissance; //sur 2 caractères
    private String numMoisNaissance; // sur 2 caractères
    private String numDepartementNaissance; // sur 2 caractères
    private String numCommuneNaissance; // sur 3 caractères
    private String numOrdreNaissance; //sur 3 caractères
    private String clef; //de sécurité/cohérence sur 2 caractères
    //en tout : 1 + 2 + 2 + 2 + 3 + 3 + 2 = 13 + 2 = 15 chiffres
    ...
}
```

```
public class Address {
    private String countryCode; // (iso 3166-1 sur 2 ou 3 caractères) (ex: "FRA" , ...)
    private String zip ;
    private String town;
    private String street;
```

```
private String number;
private String complements;
...
}
```

Reader et LineMapper java :

```
...
public class PersonWithNumAndAddressLineMapper
    implements LineMapper<PersonWithNumAndAddress>

    @Override
    public PersonWithNumAndAddress mapLine(String line, int lineNumber) throws Exception {

        var lineTokenizer = new DelimitedLineTokenizer();
        lineTokenizer.setDelimiter(";");
        lineTokenizer.setNames("id", "firstName", "lastName", "age", "active", "socialSecurityNumber", "address");

        FieldSet mainFieldSet = lineTokenizer.tokenize(line);

        var socialSecurityNumberSubLineTokenizer = new FixedLengthTokenizer();
        socialSecurityNumberSubLineTokenizer.setNames("genre", "anneeNaissance", "numMoisNaissance",
            "numDepartementNaissance", "numCommuneNaissance", "numOrdreNaissance", "clef");
        socialSecurityNumberSubLineTokenizer.setColumns(new Range(1, 1), new Range(2, 3), new Range(4, 5),
            new Range(6, 7), new Range(8, 10), new Range(11, 13), new Range(14, 15));

        BeanWrapperFieldSetMapper<SocialSecurityNumber> socialSecurityNumberFieldSetMapper
            = new BeanWrapperFieldSetMapper<SocialSecurityNumber>();
        socialSecurityNumberFieldSetMapper.setTargetType(SocialSecurityNumber.class);

        FieldSet socialSecurityNumberSubFieldSet = socialSecurityNumberSubLineTokenizer.tokenize(
            mainFieldSet.readString("socialSecurityNumber"));
        SocialSecurityNumber socialSecurityNumber =
            socialSecurityNumberFieldSetMapper.mapFieldSet(
                socialSecurityNumberSubFieldSet);

        var addressSubLineTokenizer = new DelimitedLineTokenizer();
        addressSubLineTokenizer.setDelimiter("!");
        addressSubLineTokenizer.setNames("countryCode", "zip", "town", "street", "number", "complements");

        BeanWrapperFieldSetMapper<Address> addressFieldSetMapper =
            new BeanWrapperFieldSetMapper<Address>();
        addressFieldSetMapper.setTargetType(Address.class);

        FieldSet addressSubFieldSet = addressSubLineTokenizer.tokenize(mainFieldSet.readString("address"));
        Address address = addressFieldSetMapper.mapFieldSet(addressSubFieldSet);

        PersonWithNumAndAddress personWithNumAndAddress = new PersonWithNumAndAddress();
        personWithNumAndAddress.setSocialSecurityNumber(socialSecurityNumber);
        personWithNumAndAddress.setAddress(address);
        personWithNumAndAddress.setId(mainFieldSet.readLong("id"));
        personWithNumAndAddress.setFirstName(mainFieldSet.readString("firstName"));
        personWithNumAndAddress.setLastName(mainFieldSet.readString("lastName"));
        personWithNumAndAddress.setAge(mainFieldSet.readInt("age"));
        personWithNumAndAddress.setActive(mainFieldSet.readBoolean("active"));

        return personWithNumAndAddress;
    }
}
```

à par exemple utiliser dans un "reader" configuré en java :

```
public FlatFileItemReader<PersonWithNumAndAddress>
    personCsvWithNumAndAddressFileReader() {

    var specificComplexLineMapper = new PersonWithNumAndAddressLineMapper();

    return new FlatFileItemReaderBuilder<PersonWithNumAndAddress>()
        .name("personCsvWithNumAndAddressFileReader")
        .resource(inputCsvWithNumAndAddressResource)
        .linesToSkip(1)
        .lineMapper(specificComplexLineMapper)
        .build();

    }
...

```

3.5. Gestion des fichiers json

Dans *pom.xml*

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

Exemple de fichier json :

data/input/json/inputData.json

```
[
  {"firstName":"Jean","lastName":"Bono","age":41,"active":true},
  {"firstName":"Alex","lastName":"Therieur","age":42,"active":false},
  {"firstName":"Axelle","lastName":"Aire","age":33,"active":true},
  {"firstName":"Laurent","lastName":"Houtan","age":44,"active":false}
]
```

Reader "json" configuré en xml :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:util="http://www.springframework.org/schema/util"
  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
    http://www.springframework.org/schema/util
    https://www.springframework.org/schema/util/spring-util.xsd">

  <bean name="jsonObjectReader"
    class="org.springframework.batch.item.json.JacksonJsonObjectReader">
    <constructor-arg index="0" value="tp.mySpringBatch.model.Person" />
  </bean>

  <bean id="personJsonFileItemReader"
    class="org.springframework.batch.item.json.JsonItemReader">
    <constructor-arg index="0" value="file:data/input/json/inputData.json" />
    <constructor-arg index="1" ref="jsonObjectReader" />
  </bean>
</beans>
```

Reader "json" configuré en java :

```
package tp.mySpringBatch.reader.java;
import org.springframework.batch.item.ItemReader;
import org.springframework.batch.item.json.JacksonJsonObjectReader;
import org.springframework.batch.item.json.builder.JsonItemReaderBuilder;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
import org.springframework.context.annotation.Profile;
import org.springframework.core.io.Resource;
import tp.mySpringBatch.model.Person;

@Configuration
@Profile("!xmlJobConfig")
public class MyJsonFilePersonReaderConfig {

    @Value("file:data/input/json/inputData.json") //to read in project root directory
    private Resource inputJsonResource;

    @Bean @Qualifier("json")
    ItemStreamReader<Person> personJsonFileItemReader() {
        return new JsonItemReaderBuilder<Person>()
            .name("personJsonFileItemReader")
            .jsonObjectReader(new JacksonJsonObjectReader<Person>(Person.class))
            .resource(inputJsonResource)
            .build();
    }
}
```

Writer "Json" configuré en xml :

```
<bean name="jsonObjectMarshaller"
      class="org.springframework.batch.item.json.JacksonJsonObjectMarshaller">
</bean>

<bean id="personJsonFileItemWriter"
      class="org.springframework.batch.item.json.JsonFileItemWriter">
    <constructor-arg index="0" value="file:data/output/json/outputData.json" />
    <constructor-arg index="1" ref="jsonObjectMarshaller" />
</bean>
```

Writer "Json" configuré en java :

```
package tp.mySpringBatch.writer.java;
import org.springframework.batch.item.ItemWriter;
import org.springframework.batch.item.json.JacksonJsonObjectMarshaller;
import org.springframework.batch.item.json.builder.JsonFileItemWriterBuilder;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.core.io.WritableResource;

import tp.mySpringBatch.model.Person;

@Configuration
@Profile("!xmlJobConfig")
public class MyJsonFilePersonWriterConfig {

    @Value("file:data/output/json/outputData.json") //to read in project root directory
    private WritableResource outputJsonResource;
```

```

@Bean @Qualifier("json")
ItemStreamWriter<Person> personJsonFileItemWriter() {
    return new JsonFileItemWriterBuilder<Person>()
        .name("personJsonFileItemWriter")
        .jsonObjectMarshaller(new JacksonJsonObjectMarshaller<>())
        .resource(outputJsonResource)
        .build();
}

```

Exemples de Jobs configurés en XML:

commonConfig.xml

```

...
<import resource="classpath:job/rw/jsonReader.xml" />
<import resource="classpath:job/rw/jsonWriter.xml" />
...

```

fromJsonToXmlJob.xml

```

...
<job id="fromJsonToXmlJob" xmlns="http://www.springframework.org/schema/batch">
    <step id="step1_of_fromJsonToXmlJob" >
        <tasklet>
            <chunk reader="personJsonFileItemReader" writer="personXmlFileItemWriter"
                processor="simpleUppercasePersonProcessor" commit-interval="1" />
        </tasklet>
    </step>
</job>

```

fromCsvToJsonJob.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:util="http://www.springframework.org/schema/util"
    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
        http://www.springframework.org/schema/util
        https://www.springframework.org/schema/util/spring-util.xsd">

    <job id="fromCsvToJsonJob" xmlns="http://www.springframework.org/schema/batch">
        <step id="step1_of_fromCsvToJsonJob" >
            <tasklet>
                <chunk reader="personCsvFileReader" writer="personJsonFileItemWriter"
                    processor="simpleUppercasePersonProcessor" commit-interval="1" />
            </tasklet>
        </step>
    </job>
</beans>

```


Exemples de Jobs configurés en Java:

MyBasicJsonToXmlJobConfig.java

```
package tp.mySpringBatch.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.step.builder.StepBuilder;
import org.springframework.batch.item.ItemReader;
import org.springframework.batch.item.ItemWriter;
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.mySpringBatch.listener.JobCompletionNotificationListener;
import tp.mySpringBatch.model.Person;
import tp.mySpringBatch.processor.SimpleUppercasePersonProcessor;

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

    @Bean
    public Job fromJsonToXmlJob(@Qualifier("jsonToXml") Step step1) {
        var name = "Persons jsonToXml Job";
        var jobBuilder = new JobBuilder(name, jobRepository);
        return jobBuilder.start(step1)
            .listener(new JobCompletionNotificationListener())
            .build();
    }

    @Bean @Qualifier("jsonToXml")
    public Step stepJsonToXml(@Qualifier("json") ItemReader<Person> personItemReader,
                             @Qualifier("xml") ItemWriter<Person> personItemWriter,
                             SimpleUppercasePersonProcessor
simpleUppercasePersonProcessor) {
        var name = "COPY json RECORDS To xml Step";
        var stepBuilder = new StepBuilder(name, jobRepository);
        return stepBuilder
            .<Person, Person>chunk(5, batchTxManager)
            .reader(personItemReader)
            .processor(simpleUppercasePersonProcessor)
            .writer(personItemWriter)
            .build();
    }
}
```

MyBasicCsvToJsonJobConfig.java

```
....

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

```
@Bean
public Job fromCsvToJsonJob(@Qualifier("csvToJson") Step step1) {
    var name = "Persons CsvToJson Job";
    var jobBuilder = new JobBuilder(name, jobRepository);
    return jobBuilder.start(step1)
        .listener(new JobCompletionNotificationListener())
        .build();
}

@Bean @Qualifier("csvToJson")
public Step stepCsvToJson(@Qualifier("csv") ItemReader<Person> personItemReader,
    @Qualifier("json") ItemWriter<Person> personItemWriter ,
    SimpleUppercasePersonProcessor simpleUppercasePersonProcessor) {
    var name = "CSV RECORDS To Json Step";
    var stepBuilder = new StepBuilder(name, jobRepository);
    return stepBuilder
        .<Person, Person>chunk(5, batchTxManager)
        .reader(personItemReader)
        .processor(simpleUppercasePersonProcessor)
        .writer(personItemWriter)
        .build();
}
}
```

3.6. Gestion des fichiers XML

SpringBatch manipule les fichiers XML via spring-oxm (**StaxEventItemReader** , **StaxEventItemWriter**).

Spring-oxm peut utiliser (au choix) en interne plusieurs types de "marshaller/sérialiseur" :

- **jaxb2Marshaller** (nécessitant annotations dans classe de données java
- **XStreamMarshaller** (nécessitant mapping explicite sans annotation)

Dans *pom.xml*

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-oxm</artifactId>
</dependency>
  <!-- spring oxm is for object xml mapping
       OXM is an abstract high level api .
       implementations can be based on jaxb2 (with annotations)
       or on xstream (without annotations but with mapping config)
-->

<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>
```

Exemple de fichier json :

data/input/xml/inputData.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persons>
  <person><active>true</active><age>41</age><firstName>Jean</firstName><lastName>Bono</lastName></person>
  <person><active>>false</active><age>42</age><firstName>Alex</firstName><lastName>Therieur</lastName></person>
  <person><active>true</active><age>33</age><firstName>Axelle</firstName><lastName>Aire</lastName></person>
  <person><active>>false</active><age>44</age><firstName>Olie</firstName><lastName>Condor</lastName></person>
  <person><active>true</active><age>29</age><firstName>Laurent</firstName><lastName>Houtan</lastName></person>
</persons>
```

Person.java

```
...
import jakarta.xml.bind.annotation.XmlRootElement;
....
@XmlRootElement(name = "person") //just for read/generate XML file with jaxb2 marshaller
//implements Serializable just for serialize the execution context (if needed)
public class Person implements Serializable{
    String firstName;
    String lastName;
    Integer age;
    Boolean active;
    ...
}
```

Marshaller et Reader "xml" configurés en xml :

xmlMarshaller.xml

```
<bean id="personXmlJaxb2Marshaller"
    class="org.springframework.xml.jaxb.Jaxb2Marshaller">
    <property name="classesToBeBound">
        <list>
            <value>tp.mySpringBatch.model.Person</value>
        </list>
    </property>
</bean>

<bean id="personXmlXstreamMarshaller"
    class="org.springframework.xml.xstream.XStreamMarshaller">
    <property name="typePermissions">
        <bean id="typePermission"
class="com.thoughtworks.xstream.security.ExplicitTypePermission">
            <constructor-arg>
                <list>
                    <value>tp.mySpringBatch.model.Person</value>
                </list>
            </constructor-arg>
        </bean>
    </property>
    <property name="aliases">
        <util:map>
            <entry key="person"
                value="tp.mySpringBatch.model.Person" />
            <entry key="firstName" value="java.lang.String" />
            <entry key="lastName" value="java.lang.String" />
            <entry key="age" value="java.lang.Integer" />
            <entry key="active" value="java.lang.Boolean" />
        </util:map>
    </property>
</bean>
```

xmlReader.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:util="http://www.springframework.org/schema/util"
  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
    http://www.springframework.org/schema/util
    https://www.springframework.org/schema/util/spring-util.xsd">

  <bean id="personXmlFileItemReader"
    class="org.springframework.batch.item.xml.StaxEventItemReader">
    <property name="resource" value="file:data/input/xml/inputData.xml" />
    <!-- <property name="unmarshaller" ref="personXmlJaxb2Marshaller" /> -->
    <property name="unmarshaller" ref="personXmlXstreamMarshaller" />
    <property name="fragmentRootElementName" value="person" />
  </bean>

</beans>
```

Reader "xml" configuré en java :

```
package tp.mySpringBatch.reader.java;
import org.springframework.batch.item.ItemReader;
import org.springframework.batch.item.xml.StaxEventItemReader;
import org.springframework.batch.item.xml.builder.StaxEventItemReaderBuilder;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.core.io.Resource;
import org.springframework.oxm.jaxb.Jaxb2Marshaller;
import tp.mySpringBatch.model.Person;

@Configuration
@Profile("!xmlJobConfig")
public class MyXmlFilePersonReaderConfig {

    @Value("file:data/input/xml/inputData.xml") //to read in project root directory
    private Resource inputXmlResource;

    /*
    //V1 without builder:
    @Bean @Qualifier("xml")
    ItemStreamReader<Person> personXmlFileItemReader() {

        //Create writer instance
        StaxEventItemReader<Person> itemReader =
            new StaxEventItemReader<Person>();
```

```

        var personXmlMarshaller = new Jaxb2Marshaller();
        personXmlMarshaller.setClassesToBeBound(new Class[] { Person.class });
        itemReader.setFragmentRootElementName("person");
        itemReader.setUnmarshaller(personXmlMarshaller);
        itemReader.setResource(inputXmlResource);
        return itemReader;
    }
}

//V2 with builder:
@Bean @Qualifier("xml")
ItemStreamReader<Person> personXmlFileItemReader() {

    var personXmlMarshaller = new Jaxb2Marshaller();
    personXmlMarshaller.setClassesToBeBound(new Class[] { Person.class });

    return new StaxEventItemReaderBuilder<Person>()
        .name("personXmlFileItemReader")
        .addFragmentRootElementName("person")
        .unmarshaller(personXmlMarshaller)
        .resource(inputXmlResource)
        .build();
}
}

```

Writer "XML" configuré en xml :

```

<bean id="personXmlFileItemWriter"
class="org.springframework.batch.item.xml.StaxEventItemWriter">
    <property name="resource" value="file:data/output/xml/outputData.xml" />
    <!-- <property name="marshaller" ref="personXmlJaxb2Marshaller" /> -->
    <property name="marshaller" ref="personXmlXstreamMarshaller" />
    <property name="rootTagName" value="persons" />
    <property name="overwriteOutput" value="true" />
</bean>

```

Writer "XML" configuré en java :

```

...
import org.springframework.batch.item.xml.builder.StaxEventItemWriterBuilder;
import org.springframework.xml.jaxb.Jaxb2Marshaller;
import org.springframework.xml.xstream.XStreamMarshaller;

@Configuration
@Profile("!xmlJobConfig")
public class MyXmlFilePersonWriterConfig {

    @Value("file:data/output/xml/outputData.xml") //to read in project root directory
    private WritableResource outputXmlResource;

    /*

```

```
//V1 sans builder
@Bean @Qualifier("xml")
ItemStreamWriter<Person> personXmlFileItemWriter() {

    //Create writer instance
    StaxEventItemWriter<Person> itemWriter =
        new StaxEventItemWriter<Person>();

    var personXmlMarshaller = new Jaxb2Marshaller();
    personXmlMarshaller.setClassesToBeBound(new Class[] { Person.class });

    itemWriter.setMarshaller(personXmlMarshaller);
    itemWriter.setRootTagName("persons");
    itemWriter.setResource(outputXmlResource);
    return itemWriter;
}

*/

//V2 avec builder:
@Bean @Qualifier("xml")
//@StepScope
ItemStreamWriter<Person> personXmlFileItemWriter() {

    var personXmlMarshaller = new Jaxb2Marshaller();
    personXmlMarshaller.setClassesToBeBound(new Class[] { Person.class });

    return new StaxEventItemWriterBuilder<Person>()
        .name("personXmlFileItemWriter")
        .marshaller(personXmlMarshaller)
        .rootTagName("persons")
        .resource(outputXmlResource)
        .build();
}
}
```

Exemples de Jobs configurés en XML:

commonConfig.xml

```
...
<import resource="classpath:job/rw/xmlMarshaller.xml" />
<import resource="classpath:job/rw/xmlReader.xml" />
<import resource="classpath:job/rw/xmlWriter.xml" />
...
```

fromJsonToXmlJob.xml

```
...
<job id="fromJsonToXmlJob" xmlns="http://www.springframework.org/schema/batch">
    <step id="step1_of_fromJsonToXmlJob">
        <tasklet>
            <chunk reader="personJsonFileItemReader" writer="personXmlFileItemWriter"
                processor="simpleUppercasePersonProcessor"
                commit-interval="1" />
        </tasklet>
    </step>
</job>
```

```

        </tasklet>
    </step>
</job>
...

```

fromXmlToCsvJob.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:util="http://www.springframework.org/schema/util"
    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
http://www.springframework.org/schema/util
https://www.springframework.org/schema/util/spring-util.xsd">

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

    <step id="step1_of_fromXmlToCsvJob" >
        <tasklet>
            <chunk reader="personXmlFileItemReader" writer="csvFilePersonWriter"
                processor="simpleUppercasePersonProcessor"
                commit-interval="1" />
        </tasklet>
    </step>
</job>

</beans>

```

A utiliser dans des jobs configurés en Java:

```
@Qualifier("xml") ItemWriter<Person> personItemWriter
et
```

```
@Qualifier("xml") ItemReader<Person> personItemReader
```


III - Job Parameters, Chunk jdbc , flows , ...

1. Job avec paramètres

Un job (généralement prévu pour être lancé plusieurs fois) peut heureusement être facilement paramétré de manière à faire un peu varier son comportement exact .

1.1. Lancement d'un job avec des paramètres

```
JobParameters jobParameters = new JobParametersBuilder()
    .addString("JobID", String.valueOf(System.currentTimeMillis()))
    .addString("msg1", "_my_msg1_value_")//used by PrintJobParamMessageTaskletBean and some Reader/Writer
    .addString("enableUpperCase", "true")//used by SimpleUppercasePersonProcessor
    .toJobParameters();

var jobExecution = jobLauncher.run(job, jobParameters);
var batchStatus = jobExecution.getStatus();
```

Les valeurs de ces paramètres peuvent évidemment elles mêmes provenir du **contexte externe de lancement** (propriétés systèmes , arguments de la ligne de commande , fichier de configuration,)

Exemple de lancement via un .bat :

lancer_fromCsvToXmlJob.bat

```
...
set CP=target\mySpringBatch.jar
REM set MAIN_CLASS=tp.mySpringBatch.MySpringBatchApplication
set JOB_NAME=fromCsvToXmlJob
set inputFilePath=data/input/csv/products.csv
set outputFilePath=data/output/xml/products.xml
...
REM -Dspring.profiles.active=xmlJobConfig
java -DinputFilePath=%inputFilePath% -DoutputFilePath=%outputFilePath%
    -jar %CP% %JOB_NAME%
pause
```

```
public void main_ou_run(String... args) throws Exception {
    ...
    String defaultJobName="fromCsvToXmlJob";
    String jobName = null;
    if(args.length>0) jobName=args[0];
    else
        jobName=System.getProperty("jobName", defaultJobName);

    String defaultInputFilePath="data/input/csv/products.csv";
    String inputFilePath=System.getProperty("inputFilePath", defaultInputFilePath);
    ...
}
```

```
System.out.println("****>>> jobName="+jobName + " inputFilePath="
+ inputFilePath + " outputFilePath=" + outputFilePath);
```

1.2. Prise en compte des paramètres d'un job

Attention :

```
@Value("#{jobParameters['parameterName']}")
```

ne fonctionne qu'au sein d'un **Bean d'exécution** ayant le **scope** "step" ou "job" .

NB : un **bean d'exécution** (avec **@StepScope** ou bien **@JobScope**) peut être un **@Component** de type "tasklet" ou un "processor" ou encore un "reader" ou "writer" mais ne peut pas être un bean de configuration (pas un **Job** , pas un **Step** , ...)

Au niveau d'un processeur :

```
package tp.mySpringBatch.processor;

import org.springframework.batch.core.configuration.annotation.JobScope;
import org.springframework.batch.core.configuration.annotation.StepScope;
import org.springframework.batch.item.ItemProcessor;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
import tp.mySpringBatch.model.Person;

@Component
@StepScope
//@JobScope
public class SimpleUppercasePersonProcessor implements ItemProcessor<Person,Person>{

    @Value("#{jobParameters['enableUpperCase']}")
    private Boolean enableUpperCase=true;

    @Override
    public Person process(Person pers) throws Exception {
        String lastName =
            enableUpperCase?pers.getLastName().toUpperCase():pers.getLastName();
        Person person=new Person(pers.getFirstName(),lastName,pers.getAge(),pers.getActive());
        return person;
    }
}
```

Le nom de la personne est ici transformé en majuscules si **enableUpperCase** est fixé à true .

Au niveau d'un tasklet codé en java :

```
...
@Component
@StepScope
//@JobScope
public class PrintJobParamMessageTaskletBean extends PrintMessageTasklet{

    @Value("#{jobParameters['msgI']}")
    public void setMessage(String message) {
        super.setMessage(message);
    }
}
```

```
}
}
```

→ affiche (via code hérité) la valeur du paramètre 'msg1' (exemple : `_my_msg1_value_`)

Au niveau d'un tasklet paramétré en XML :

```
...
<bean class="org.springframework.batch.core.scope.StepScope" />
<!-- it is necessary for scope="step"
but may need spring.main.allow-bean-definition-overriding=true in application.properties -->

<bean id="printMessageTaskletMsg1" scope="step"
      class="tp.mySpringBatch.tasklet.PrintMessageTasklet

```

Au niveau d'un reader ou d'un writer:

```
@Configuration
public class MyXyWriterConfig {

    @Bean @StepScope
    FlatFileItemWriter<Person> xyWriter(
        @Value("#{jobParameters['paramName']}") String param) {
        ...
    }
}
```

1.3. Validation et utilisation des "jobParameters"

Pour un paramètre de type "relativePathNameOfFile" (récupéré au format "String") , on pourra facilement transformer cela en une ressource (de type **Resource** ou **WritableResource**) avec un comportement identique à `@Value("file:relativePathNameOfFile")` via le constructeur de `org.springframework.core.io.FileSystemResource()` .

Exemple :

```
@Value("#{jobParameters['outputCsvPath']}")
private String outputCsvPath ;
...

WritableResource outputCsvResource = new FileSystemResource(outputCsvPath);
```

Certaines méthodes bien pratiques telles que `.exists()` , `isReadable()` , `isWritable()` sont disponibles sur les interfaces **Resource** et **WritableResource** .

...

1.4. 2 jobInstances must at least have a different jobParameter



<https://blog.csdn.net/topdeveloper1>

```
JobParametersBuilder()  
    .addLong("timeStampOfJobInstance", System.currentTimeMillis())  
    // .addString("paramName", "paramValue")  
    .toJobParameters();
```

Un paramètre toujours changeant tel que *timeStampOfJobInstance* est nécessaire pour pouvoir lancer plusieurs instances d'un même job (Deux "jobInstances" ne peuvent pas avoir tous les "jobParameters" identiques ; il faut qu'il y ait au moins une différence) .

2. Processeurs (filtrage, composition, ...)

2.1. Filtrage de données via un processeur

Exemple :

```
public class FilterPersonByAgeProcessor implements ItemProcessor<Person,Person>{
    @Override
    public Person process(Person item) throws Exception {
        if (item.getAge()<18)
            return null;
        else
            return item;
    }
}
```

Il suffit de retourner **null** au niveau d'un processeur intermédiaire pour qu'un élément lu par le "reader" ne soit pas retransmis et donc pas traité par le "writer" .

2.2. Enchaînement de processeurs

Appeler plusieurs fois de suite .processor(aProcessor) sur stepBuilder ne permet malheureusement pas d'enregistrer plusieurs processeurs complémentaires (un appel ultérieur à .processor() remplace la valeur de l'appel précédent).

On peut par contre mettre en place facilement un enchaînement de processeurs via un "processeur composite" :

```
@Bean @StepScope @Qualifier("upperCaseLastname_ageFiltering")
public CompositeItemProcessor<Person,Person>
    myUppercaseLastnameAgeFilteringCompositeProcessor(
        FilterPersonByAgeProcessor filterPersonByAgeProcessor,
        SimpleUppercasePersonProcessor simpleUppercasePersonProcessor
    ) {
    CompositeItemProcessor<Person,Person> processor = new CompositeItemProcessor<>();
    processor.setDelegates(Arrays.asList(filterPersonByAgeProcessor,
                                         simpleUppercasePersonProcessor));
    return processor;
}
```

```
@Bean @Qualifier("csvToXml")
public Step stepXyz(...,
    @Qualifier("upperCaseLastname_ageFiltering") CompositeItemProcessor compositeProcessor)
{
    ...
    return stepBuilder...
        .processor(compositeProcessor)...
        .build();
}
```

3. Gestion des bases de données

Principaux "Reader" et "Writer" pour les bases de données relationnelles :

JdbcCursorItemReader	Lecture des lignes 1 à 1 via un curseur JDBC
JdbcPagingItemReader	Lecture via JDBC et en mode "par page"
...	
JdbcBatchItemWriter	Ecriture via jdbc et NamedParameterJdbcTemplate
JpaPagingItemReader	Ecriture via Jpa (attention aux performances)

PersonRowMapper.java (utilisé par **JdbcCursorItemReader** ou **JdbcPagingItemReader**)

```
package tp.mySpringBatch.db;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;
import tp.mySpringBatch.model.Person;

public class PersonRowMapper implements RowMapper<Person> {

    public static final String FIRSTNAME_COLUMN = "first_name";
    public static final String LASTNAME_COLUMN = "last_name";
    public static final String AGE_COLUMN = "age";
    public static final String ACTIVE_COLUMN = "is_active";

    @Override
    public Person mapRow(ResultSet rs, int rowNum) throws SQLException {
        Person person = new Person();

        person.setFirstName(rs.getString(FIRSTNAME_COLUMN));
        person.setLastName(rs.getString(LASTNAME_COLUMN));
        person.setAge(rs.getInt(AGE_COLUMN));
        person.setActive(rs.getBoolean(ACTIVE_COLUMN));

        return person;
    }
}
```

JdbcReader configuré en XML :

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:util="http://www.springframework.org/schema/util"
    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
http://www.springframework.org/schema/util
https://www.springframework.org/schema/util/spring-util.xsd">

```

<!--

NB: la table person doit exister

le inputdbDataSource provient actuellement de la configuration java

-->

```

<bean id="jdbcPersonReader"
      class="org.springframework.batch.item.database.JdbcCursorItemReader">
  <property name="dataSource" ref="inputdbDataSource" /> <!-- from java config -->
  <property name="sql" value="select first_name, last_name, age, is_active from person" />
  <property name="rowMapper" >
    <bean class="tp.mySpringBatch.db.PersonRowMapper">
    </bean>
  </property>
</bean>
</beans>

```

JdbcWriter configuré en XML :

```

<bean id="jdbcItemWriter"
      class="org.springframework.batch.item.database.JdbcBatchItemWriter">
  <property name="dataSource" ref="outputdbDataSource" /> <!-- from java config -->
  <property name="sql" value=
    "insert into person (first_name, last_name, age, is_active) values (:firstName,:lastName,:age,:active)" />
  <property name="itemSqlParameterSourceProvider" >
    <bean class="org.springframework.batch.item.database.BeanPropertyItemSqlParameterSourceProvider">
    </bean>
  </property>
</bean>

```

Exemples de Jobs configurés en XML:

commonConfig.xml

```

...
<import resource="classpath:job/rw/jdbcReader.xml" />
<import resource="classpath:job/rw/jdbcWriter.xml" />
...

```

insertIntoDbFromCsvJob.xml

```

...
<job id="insertIntoDbFromCsvJob" xmlns="http://www.springframework.org/schema/batch">
  <step id="step1_of_insertIntoDbFromCsvJob" >
    <tasklet>
      <chunk reader="personCsvFileReader" writer="jdbcItemWriter" commit-interval="1" />
    </tasklet>
  </step>
</job>

```

```

    </tasklet>
  </step>
</job>

```

fromCsvToJsonJob.xml

```

<job id="insertIntoCsvFromDbJob" xmlns="http://www.springframework.org/schema/batch">
  <step id="step1_of_insertIntoCsvFromDbJob" >
    <tasklet>
      <chunk reader="jdbcPersonReader" writer="csvFilePersonWriter" commit-interval="1" />
    </tasklet>
  </step>
</job>

```

JdbcReader configuré en Java :

```

...
import org.springframework.batch.item.database.builder.JdbcCursorItemReaderBuilder;
...
@Configuration
@Profile("!xmlJobConfig")
public class MyDbPersonReaderConfig {

    private static final String SELECT_QUERY =
        "SELECT id, first_name, last_name, age, is_active FROM person";

    @Bean @Qualifier("db")
    ItemReader<Person> jdbcPersonReader(
        @Qualifier("inputdb") DataSource inputdbDataSource) {
        return new JdbcCursorItemReaderBuilder<Person>()
            .name("jdbcPersonReader")
            .dataSource(inputdbDataSource)
            .sql(SELECT_QUERY)
            .rowMapper(new PersonRowMapper())
            .build();
    }
}

```


JdbcWriter configuré en Java:

```

....
import org.springframework.batch.item.database.BeanPropertyItemSqlParameterSourceProvider;
import org.springframework.batch.item.database.JdbcBatchItemWriter;
import org.springframework.batch.item.database.builder.JdbcBatchItemWriterBuilder;
...
@Configuration
@Profile("!xmlJobConfig")
public class MyDbPersonWriterConfig {

    private static final String INSERT_QUERY = ""
        insert into person (first_name, last_name, age, is_active)
        values (:firstName,:lastName,:age,:active)"";

    @Bean @Qualifier("db")
    public JdbcBatchItemWriter<Person> jdbcItemWriter(
        @Qualifier("outputdb") DataSource outputdbDataSource) {
        return new JdbcBatchItemWriterBuilder<Person>()
            .itemSqlParameterSourceProvider(
                new BeanPropertyItemSqlParameterSourceProvider<Person>()
            )
            .dataSource(outputdbDataSource)
            .sql(INSERT_QUERY)
            .build();
    }
}

```

Exemples de Jobs configurés en Java:*CsvToDatabaseJobConfig.java*

```

....
@Configuration
@Profile("!xmlJobConfig")
public class CsvToDatabaseJobConfig extends MyAbstractJobConfig{
    ...
    @Bean @Qualifier("csvToDb")
    public Step stepCsvToDb(@Qualifier("csv") ItemReader<Person> reader,
        @Qualifier("db") ItemWriter<Person> writer) {
        var name = "INSERT CSV RECORDS To DB Step";
        var builder = new StepBuilder(name, jobRepository);
        return builder
            .<Person, Person>chunk(5, batchTxManager)
            .reader(reader)
            .writer(writer)
            .build();
    }
}

```

4. "Step flow" / configuration

4.1. Enchaînements séquentiels

Via un paramétrage XML :

```
<bean id="printMessageWithDelayTaskletB"
      class="tp.mySpringBatch.tasklet.PrintMessageWithDelayTasklet">
  <property name="message" value="BBBB"/>
  <property name="delay" value="2000"/>
</bean>
...
<job id="mySimpleSequentialStepsJob" xmlns="http://www.springframework.org/schema/batch">

  <step id="step1_of_mySimpleSequentialStepsJob"
        next="step2_of_mySimpleSequentialStepsJob">
    <tasklet ref="printMessageWithDelayTaskletA" />
  </step>

  <step id="step2_of_mySimpleSequentialStepsJob"
        next="step3_of_mySimpleSequentialStepsJob">
    <tasklet ref="printMessageWithDelayTaskletB" />
  </step>

  <step id="step3_of_mySimpleSequentialStepsJob">
    <tasklet ref="printMessageWithDelayTaskletC" />
  </step>
</job>
```

Via un paramétrage Java :

```
...
@Bean(name="mySimpleSequentialStepsJob")
public Job mySimpleSequentialStepsJob() {
  var name = "mySimpleSequentialStepsJob";
  var stepBuilder = new StepBuilder(name, jobRepository);

  Step step1 = stepBuilder.tasklet(new PrintMessageWithDelayTasklet("from_step_1",2000L),
                                  this.batchTxManager).build();
  // idem pour step2 et step3 ...

  var jobBuilder = new JobBuilder(name, jobRepository);
  return jobBuilder
    .start(step1)
    .next(step2)
    .next(step3)
    .listener(new JobCompletionNotificationListener())
    .build();
}
```

4.2. Enchaînements conditionnels simples

Via un paramétrage XML :

```
<bean id="printMessageWithDelayTaskletA"
class="tp.mySpringBatch.tasklet.PrintMessageWithDelayTasklet">
  <!-- <property name="message" value="AAAA_OK"/> -->
  <property name="message" value="AAAA_ERROR"/>
  <property name="delay" value="2000"/>
</bean>
...
<job id="mySimpleConditionalStepsJob" xmlns="http://www.springframework.org/schema/batch">
  <step id="step1_of_mySimpleConditionalStepsJob" >
    <tasklet ref="printMessageWithDelayTaskletA" />
    <next on="FAILED" to="step3_of_mySimpleConditionalStepsJob" />
    <!-- <next on="COMPLETED" to="step2_of_mySimpleConditionalStepsJob" /> -->
    <next on="*" to="step2_of_mySimpleConditionalStepsJob" />
  </step>

  <step id="step2_of_mySimpleConditionalStepsJob"
    next="step3_of_mySimpleConditionalStepsJob">
    <tasklet ref="printMessageWithDelayTaskletB" />
  </step>

  <step id="step3_of_mySimpleConditionalStepsJob" >
    <tasklet ref="printMessageWithDelayTaskletC" />
  </step>
</job>
```

Via un paramétrage Java :

```
@Bean(name="mySimpleConditionalStepsJob")
public Job mySimpleConditionalStepsJob() {
  ...
  //Step step1 = stepBuilder.tasklet(new PrintMessageWithDelayTasklet("from_step_1_ok",2000L), ...).build();
  Step step1 = stepBuilder.tasklet(new PrintMessageWithDelayTasklet("from_step_1_error",2000L),
                                  this.batchTxManager).build();
  //idem pour step2, step3 ...
  var jobBuilder = new JobBuilder(name, jobRepository);
  return jobBuilder
    .start(step1).on("FAILED").to(step3)
    .from(step1).on("*").to(step2)
    .from(step2).on("*").to(step3)
    .end()
    .listener(new JobCompletionNotificationListener())
    .build();
}
```

Comportement de l'exemple précédent :

- si l'exécution de "step1" retourne "FAILED" on enchaîne alors directement "step3"
- si l'exécution de "step1" retourne une autre valeur on enchaîne "step2" puis "step3"

NB : Les valeurs à comparer (du côté `on="OK"` ou `on="FAILED"`) correspondent à une des valeurs possibles de `ExitStatus`. En plus des constantes prédéfinies `ExitStatus.FAILED`, ..., `ExitStatus.COMPLETED` on peut retourner n'importe quelle chaîne de caractères (ex : "OK") via `.setExitStatus(new ExitStatus("OK_ou_autre"))`;

```
public class PrintMessageWithDelayTasklet implements Tasklet{
    private String message;
    private Long delay=1000L; //ms (1000=default value)

    public PrintMessageWithDelayTasklet(String message, Long delay) {
        super(); this.message = message; this.delay = delay;
    }

    public PrintMessageWithDelayTasklet() { super(); }

    @Override
    public RepeatStatus execute(StepContribution contribution, ChunkContext chunkContext)
        throws Exception {
        Thread.sleep(delay);
        System.out.println(message);
        if(message==null || message.toLowerCase().contains("error"))
            contribution.setExitStatus(ExitStatus.FAILED);
        else
            contribution.setExitStatus(ExitStatus.COMPLETED);
        //NB: if error detected in processor : this.stepExecution.setExitStatus(ExitStatus.FAILED);
        return RepeatStatus.FINISHED;
    }
    // plus get/set ...
}
```

Quelques idées de "ExitStatus" personnalisés :

- **"COMPLETED_WITH_SKIPS"** : si step bien terminé mais avec quelques erreurs non bloquantes sautées (via `StepExecutionListener.afterStep(stepExecution)` et en testant si `stepExecution.getSkipCount()` est strictement supérieur à 0).
- "..."

4.3. Contrôle du "ExitStatus" via un Listener

Dans certains cas, on peut (juste après l'exécution d'un "Step") analyser certains états pour décider la valeur à donner au "ExitStatus" . On peut pour cela s'appuyer sur un "decider" ou un "listener" :

Exemple :

```
public class SkipCheckingListener extends StepExecutionListenerSupport {
    public ExitStatus afterStep(StepExecution stepExecution) {
        String exitCode = stepExecution.getExitStatus().getExitCode();
        if (!exitCode.equals(ExitStatus.FAILED.getExitCode()) &&
            stepExecution.getSkipCount() > 0) {
            return new ExitStatus("COMPLETED WITH SKIPS");
        }
        else {
            return null;
        }
    }
}
```

4.4. Enchaînements conditionnels avec "decider"

Une **décision** est la **prise en compte d'une condition logique de branchement** dans un **graphe d'activités** (UML ou BPMN ou autre) **qui sert à orienter le "workflow"** dans un sens ou un autre.

Au sein de SpringBatch , une **"decision"** peut être vue comme une "pseudo-step" n'exécutant pas de traitement mais analysant une situation au niveau du Job et retournant une valeur qui sera testée pour contrôler l'enchaînement des prochains "step" .

Ça se code comme une classe java implémentant l'interface **JobExecutionDecider** comme dans l'exemple ci-après :

MySkipCheckingDecider.java

```
package tp.mySpringBatch.decider;
import org.springframework.batch.core.ExitStatus;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.StepExecution;
import org.springframework.batch.core.job.flow.FlowExecutionStatus;
import org.springframework.batch.core.job.flow.JobExecutionDecider;

public class MySkipCheckingDecider implements JobExecutionDecider {

    @Override
    public FlowExecutionStatus decide(JobExecution jobExecution, StepExecution stepExecution) {
        //stepExecution as "lastStepExecution" (may be null)
        if(stepExecution==null)
            return new FlowExecutionStatus(ExitStatus.UNKNOWN.toString());
        if(!ExitStatus.FAILED.equals(stepExecution.getExitStatus())
            && stepExecution.getSkipCount()>0)
```

```

        return new FlowExecutionStatus("COMPLETED_WITH_SKIPS");
    else
        return new FlowExecutionStatus(stepExecution.getExitStatus().getExitCode().toString());
    }
}

```

NB : via le mode "faultTolerant() / skip" un "step" peut ignorer certaines erreurs non bloquantes (ex : un petit nombre de lignes mal structurées dans un fichier CSV) . Dans ce cas là le skipCount sera supérieur à 0 et de "decider" précédent retournera "**COMPLETED_WITH_SKIPS**" (plus précis que "COMPLETED")

Utilisation en XML :

```

<bean id="mySkipCheckingDecider"
      class="tp.mySpringBatch.decider.MySkipCheckingDecider" />

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

  <step id="step1_of_withDecisionFlowJob" next="mySkipCheckingDecision">
    <tasklet>
      <chunk reader="personCsvFileReader or personCsvWithErrorsFileReader"
              writer="personJsonFileItemWriter"
              processor="simpleUppercasePersonProcessor"
              commit-interval="1" skip-limit="5" >
        <skippable-exception-classes>
          <include class="org.springframework.batch.item.ItemReaderException" />
        </skippable-exception-classes>
      </chunk>
    </tasklet>
  </step>
  <decision id="mySkipCheckingDecision" decider="mySkipCheckingDecider">
    <next on="COMPLETED_WITH_SKIPS" to="step3_of_withDecisionFlowJob" />
    <next on="COMPLETED" to="step2_of_withDecisionFlowJob" />
  </decision>

  ... step2, step3, ...
</job>

```

Autres possibilités :

```

<decision id="mySkipCheckingDecision" decider="mySkipCheckingDecider">
  <end on="xxx" />

  <next on="yyyy" to="stepyyyy" />

  <fail on="*" />
</decision>

```

Utilisation en JAVA :

```

...
@Bean
public MySkipCheckingDecider mySkipCheckingDecider() {
    return new MySkipCheckingDecider();
}

@Bean(name="withDecisionFlowJob")
public Job withDecisionFlowJob(
    MySkipCheckingDecider mySkipCheckingDecider,
    @Qualifier("csvWithSkipsErrorsToJson") Step step1
    // @Qualifier("csvToJson") Step step1
) {
    var name = "withDecisionFlowJob";
    var stepBuilder = new StepBuilder(name, jobRepository);

    Step stepWithSkips = stepBuilder.tasklet(new
        PrintMessageWithDelayTasklet("COMPLETED_WITH_SKIPS",500L),
        this.batchTxManager).build();
    Step stepWithoutSkip = stepBuilder.tasklet(new
        PrintMessageWithDelayTasklet("COMPLETED",500L), this.batchTxManager).build();

    var jobBuilder = new JobBuilder(name, jobRepository);

    final Flow withOrWithoutSkipFlow =
        new FlowBuilder<SimpleFlow>("withOrWithoutSkipFlow")
        .start(step1)
        .next(mySkipCheckingDecider).on("COMPLETED_WITH_SKIPS").to(stepWithSkips)
        .from(mySkipCheckingDecider).on("COMPLETED").to(stepWithoutSkip)
        .end();//end of flow

    return jobBuilder
        .start(withOrWithoutSkipFlow)
        .end();//end FlowBuilder and return to JobBuilder
        .listener(new JobCompletionNotificationListener())
        .build();
}

```

Autres possibilités (variantes):

from_or_next(stepXy).on("FAILED").end()
 to ending full job with *COMPLETED* status when intermediate stepXy "FAILED"

from_or_next(stepXy).on("FAILED").fail()
 to ending full job with *FAILED* status when intermediate stepXy "FAILED"

4.5. Quelques autres possibilités

Réutilisation de Flow :

```
<flow id="myFlow" >
  <step id="stepA" next="stepB"> ....</step>
  <step id="stepB" next="stepC"> ....</step>
  <step id="stepC" > ....</step>
</flow>
```

```
<job id=myJob">
  <flow parent="myFlow" id="mySteps" next="stepD">
    <step id="stepD"> .... </step>
  </job>
```

Réutilisation de Job :

```
<job id="job1" >
  <step id="stepA" next="stepB"> ....</step>
  <step id="stepB" next="stepC"> ....</step>
  <step id="stepC" > ....</step>
</job>
```

```
<job id=job2">
  <step id="stepsOfJob1" next="stepD">
    <job ref="job1" />
  </step>
  <step id="stepD"> .... </step>
</job>
```

Passer des infos (ou données) entre différents "Step" :

1) Au sein d'un "Reader" ou "Writer" personnalisé , on peut sauvegarder (dans un attribut "private") le stepExecution dont une référence est passée via @BeforeStep saveStepExecution(...)

Au sein d'un Tasklet implémentant StepExecutionListener on peut faire de même au sein de beforeStep(...)

2) On peut ensuite accéder au contexte du Step (ou bien du Job) via
stepExecution.getExecutionContext() ou bien
stepExecution.getJobExecution().getExecutionContext()

3) un **ExecutionContext** est une Map (avec méthodes traditionnelles **.put(key,value)** et **.get(key)**) dans laquelle on peut stocker et récupérer certaines valeurs.

4.6. Enchaînements en parallèle via des partitions

Exemple de "partitioner" simple :

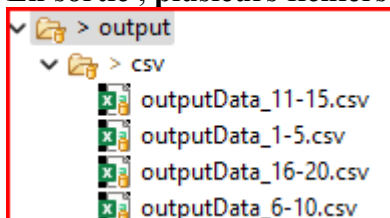
En entrée , une **table** avec n enregistrements numérotés de 1 à N(ici=20) :

PERSON_ID	FIRST_NAME	LAST_NAME	AGE	IS_ACTIVE
1	Jean	Bono	40	TRUE
2	Laurent	Houtan	30	FALSE
3	Alain	Verse	20	TRUE
4	Olie	Condor	35	FALSE
5	Alex	Therieur	28	TRUE
6	prenom6	nom6	33	TRUE
7	prenom7	nom7	33	TRUE

....

20 , prenom20, nom20,33,true

En sortie , plusieurs fichiers ".csv" comportant chacun une plage d'enregistrement :



Contenu de outputData_16-20.csv :

```
firstname;lastname;age;active
prenom16;nom16;33;true
prenom17;nom17;33;true
prenom18;nom18;33;true
prenom19;nom19;33;true
prenom20;nom20;33;true
```

NB1 : chaque plage d'enregistrements (partition) sera de taille range=5 et sera traitée par un thread dédié (nbThreads = gridSize=4).

Un **managerStep** maître va créer et lancer plusieurs "**workerStep**" (géré par un thread spécifique) . Chacun de ces "**workerStep**" (qui vont s'exécuter en parallèle) va extraire une plage d'enregistrements dans la table , effectuer une éventuelle transformation et générer un des fichiers .csv en sortie .

NB2: de manière à paramétrer le travail de chacun des "workerStep" , on a besoin de coder un objet de type "**Partitioner**" qui va retourner une Map entre **nomDePartition** et sousMap (de type **ExecutionContext**) . Chacun des "**ExecutionContext**" sera un **paquet de paramètres** qui sera utilisé par un "workerStep" pour traiter une partition.

NB3: chaque "**workerStep**" va utiliser des "**reader**" et "**writer**" en scope="test" qui vont se baser sur les paramètres (ici "fromId" et "toId") d'un objet "ExecutionContext" préalablement créé par le "partitioner".

MyRangePartitioner.java

```

package tp.mySpringBatch.partitioner;

import java.util.HashMap; import java.util.Map;
import org.slf4j.Logger; import org.slf4j.LoggerFactory;
import org.springframework.batch.core.partition.support.Partitioner;
import org.springframework.batch.item.ExecutionContext;

public class MyRangePartitioner implements Partitioner {
    private static Logger logger = LoggerFactory.getLogger(MyRangePartitioner.class);

    //range (as rangeSize) is the number of entries that will be managed
    //by a thread/partition_executionContext
    private Integer range=10; //default value

    public MyRangePartitioner(Integer range) { super(); this.range = range; }
    public MyRangePartitioner() { super(); }

    @Override
    public Map<String, ExecutionContext> partition(int gridSize) {
        //NB: grid size will be the number of threads (in //)

        Map<String, ExecutionContext> partitionMap =
            new HashMap<String, ExecutionContext>();
        int fromId = 1;
        int toId = range;
        for (int i = 1; i <= gridSize; i++) {
            //NB: a springBatch ExecutionContext is a sort of Map of any key/value
            //that will be used by a executionThread
            ExecutionContext partitionExecutionContext = new ExecutionContext();
            partitionExecutionContext.putString("name", "partition_" + i);

            //first id value in table to be managed by this thread/partition
            partitionExecutionContext.putInt("fromId", fromId);

            //last id value in table to be managed by this thread/partition
            partitionExecutionContext.putInt("toId", toId);
            logger.debug(partitionExecutionContext.getString("name") +
                " managed by a specific thread" + " will be use to manage data whith id between "
                + partitionExecutionContext.getInt("fromId")
                + "and " + partitionExecutionContext.getInt("toId"));

            partitionMap.put(partitionExecutionContext.getString("name"),
                partitionExecutionContext);

            fromId = toId + 1;
            toId += range;
        }
        return partitionMap;
    }

    public Integer getRange() { return range; }
    public void setRange(Integer range) { this.range = range; }
}

```

Configuration en 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="myRangePartitioner" class="tp.mySpringBatch.partitionner.MyRangePartitioner" >
    <property name="range" value="5" />
    <!-- interval size of index range to be managed by a specific thread/partition -->
  </bean>

  <bean id="taskExecutor" class="org.springframework.core.task.SimpleAsyncTaskExecutor" />

  <!-- partitioner job -->
  <job id="myPartitionJob" xmlns="http://www.springframework.org/schema/batch">

    <!-- master/manager step, 4 threads (grid-size) -->
    <step id="managerStep">
      <partition step="workerStep" partitioner="myRangePartitioner">
        <handler grid-size="4" task-executor="taskExecutor" />
      </partition>
    </step>

  </job>

  <!-- each thread will run this job, with different stepExecutionContext values. -->
  <step id="workerStep" xmlns="http://www.springframework.org/schema/batch">
    <tasklet>
      <chunk reader="jdbcPartitionPersonReader" writer="csvFilePartitionPersonWriter"
        commit-interval="1" processor="withDelayAndThreadNameLogPersonProcessor" />
    </tasklet>
  </step>

  <!--
  NB: jdbcPartitionPersonReader (of scope="step") is defined in job/rw/jdbcReader
    and it will read records with person_id from #{stepExecutionContext[fromId]}
        to #{stepExecutionContext[toId]}

  NB: csvFilePartitionPersonWriter (of scope="step") is defined in job/rw/csvWriter
    and it will generate output csv file with name=
    "file:data/output/csv/outputData_#{stepExecutionContext[fromId]}-#{stepExecutionContext[toId]}.csv"
  -->
</beans>

```

Configuration en Java :

MyPartitionJobConfig.java

```
package tp.mySpringBatch.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.partition.support.Partitioner;
import org.springframework.batch.core.step.builder.StepBuilder;
import org.springframework.batch.item.ItemReader;
import org.springframework.batch.item.ItemWriter;
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 org.springframework.core.task.TaskExecutor;
import tp.mySpringBatch.listener.JobCompletionNotificationListener;
import tp.mySpringBatch.model.Person;
import tp.mySpringBatch.partitioner.MyRangePartitioner;

```

@Configuration

@Profile("!xmlJobConfig")

public class *MyPartitionJobConfig* extends MyAbstractJobConfig{

 @Bean(name="myPartitionJob")

```

public Job myPartitionJob(@Qualifier("managerDbToCsv") Step managerStep) {
    var builder = new JobBuilder("myPartitionJob_DbToCsv", jobRepository);
    return builder.start(managerStep).listener(new JobCompletionNotificationListener()).build();
}

```

 @Bean

```

public Partitioner myPartitioner() {
    return new MyRangePartitioner(5); //rangeSize=5
}

```

 @Bean

```

public TaskExecutor myTaskExecutor() {
    return new org.springframework.core.task.SimpleAsyncTaskExecutor();
}

```

 @Bean @Qualifier("managerDbToCsv")

```

public Step managerStep(
    @Qualifier("myTaskExecutor")TaskExecutor taskExecutor,
    @Qualifier("myPartitioner") Partitioner partitioner,
    @Qualifier("workerDbToCsv") Step workerStep) {
    var builder = new StepBuilder("managerStep", jobRepository);
    return builder
        .partitioner("workerStep", partitioner)
        .step(workerStep)
        .gridSize(4)
        .taskExecutor(taskExecutor)
        .build();
}

```

 @Bean @Qualifier("workerDbToCsv")

```

public Step workStepDbToCsv(@Qualifier("db_ withPartition") ItemReader<Person> reader,
    @Qualifier("csv withPartition") ItemWriter<Person> writer) {
    var name = "Extract CSV RECORDS From DB Step With partition";
    var builder = new StepBuilder(name, jobRepository);
    return builder
        .<Person, Person>chunk(1, batchTxManager)

```

```

        .reader(reader)
        .writer(writer)
        .build();
    }
}

```

Reader avec partition (ex XML) :

```

<bean id="jdbcPartitionPersonReader" scope="step"
class="org.springframework.batch.item.database.JdbcPagingItemReader">
  <property name="dataSource" ref="inputdbDataSource" /> <!-- from java config -->
  <property name="queryProvider">
    <bean
      class="org.springframework.batch.item.database.support.SqlPagingQueryProviderFactoryBean">
        <property name="dataSource" ref="inputdbDataSource" />
        <property name="selectClause" value="select person_id, first_name, last_name, age, is_active" />
        <property name="fromClause" value="from person" />
        <property name="whereClause"
          value="where person_id >= :fromId and person_id <= :toId" />
        <property name="sortKey" value="person_id" />
      </bean>
    </property>
    <property name="parameterValues">
      <map>
        <entry key="fromId" value="#{stepExecutionContext[fromId]}" />
        <entry key="toId" value="#{stepExecutionContext[toId]}" />
      </map>
    </property>
    <property name="pageSize" value="5" />
    <property name="rowMapper">
      <bean class="tp.mySpringBatch.db.PersonRowMapper">
      </bean>
    </property>
  </bean>

```

Reader avec partition (ex Java) :

```

package tp.mySpringBatch.reader.java;

import org.springframework.batch.core.configuration.annotation.StepScope;
import org.springframework.batch.item.database.PagingQueryProvider;
import org.springframework.batch.item.database.builder.JdbcPagingItemReaderBuilder;
import org.springframework.batch.item.database.support.SqlPagingQueryProviderFactoryBean;
...

@Configuration
@Profile("!xmlJobConfig")
public class MyDbPersonReaderWithPartitionConfig {

    @Bean @Qualifier("db_withPartition")
    @StepScope

```

```

ItemReader<Person> jdbcPartitionPersonReader(
    @Qualifier("inputdb") DataSource inputdbDataSource,
    @Value("#{stepExecutionContext[fromId]}") String fromId,
    @Value("#{stepExecutionContext[toId]}") String toId
) throws Exception {

    SqlPagingQueryProviderFactoryBean pagingQueryProviderFactory =
        new SqlPagingQueryProviderFactoryBean();
    pagingQueryProviderFactory.setDataSource(inputdbDataSource);
    pagingQueryProviderFactory.setSelectClause(
        "select person_id, first_name, last_name, age, is_active");
    pagingQueryProviderFactory.setFromClause("from person");
    pagingQueryProviderFactory.setWhereClause(
        "where person_id >= :fromId and person_id <= :toId");
    pagingQueryProviderFactory.setSortKey("person_id");
    PagingQueryProvider pagingQueryProvider=pagingQueryProviderFactory.getObject();

    Map<String,Object> parameterValues = new HashMap<>();
    parameterValues.put("fromId", fromId);
    parameterValues.put("toId", toId);

    return new JdbcPagingItemReaderBuilder<Person>()
        .name("jdbcPartitionPersonReader")
        .dataSource(inputdbDataSource)
        .queryProvider(pagingQueryProvider)
        .parameterValues(parameterValues)
        .pageSize(5)
        .rowMapper(new PersonRowMapper())
        .build();
}

```

Writer avec partition (ex XML) :

```

<bean id="csvFilePartitionPersonWriter" scope="step"
    class="org.springframework.batch.item.file.FlatFileItemWriter">
    <property name="resource"
value="file:data/output/csv/outputData_#{@stepExecutionContext[fromId]}-
#{@stepExecutionContext[toId]}.csv" />
    <property name="appendAllowed" value="false" />
    <property name="lineAggregator">
        <bean class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
            <property name="delimiter" value=";" />
            <property name="fieldExtractor">
                <bean
                    class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor">
                        <property name="names" value="firstName,lastName,age,active" />
                    </bean>
                </property>
            </bean>
        </property>
    </bean>
    </property>

```

```
</bean>
```

Writer avec partition (ex Java) :

```
package tp.mySpringBatch.writer.java;

import org.springframework.batch.core.configuration.annotation.StepScope;
import org.springframework.batch.item.file.FlatFileItemWriter;
import org.springframework.batch.item.file.builder.FlatFileItemWriterBuilder;
...
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.WritableResource;

@Configuration
@Profile("!xmlJobConfig")
public class MyCsvFilePersonWriterWithPartitionConfig {

    @Bean @StepScope @Qualifier("csv_withPartition")
    FlatFileItemWriter<Person> csvFilePartitionPersonWriter(
        @Value("#{stepExecutionContext[fromId]}") String fromId,
        @Value("#{stepExecutionContext[toId]}") String toId) {

        WritableResource outputPartitionCsvResource = new
            FileSystemResource("data/output/csv/outputData_" + fromId + "-" + toId + ".csv");

        return new FlatFileItemWriterBuilder<Person>()
            .name("csvFilePersonWriter")
            .resource(outputPartitionCsvResource)
            .delimited()
            .delimiter(";")
            .names("firstName", "lastName", "age", "active")
            .headerCallback((writer)-> {writer.write("firstname;lastname;age;active");})
            .build();
    }
}
```

NB: pour bien visualiser l'aspect "multi-thread" et le gain en vitesse d'exécution on pourra (en TP) s'appuyer un processeur effectuant volontairement une petite pause :

```
@Component @StepScope
public class WithDelayAndThreadNameLogPersonProcessor implements
ItemProcessor<Person,Person>{

    ...
    @Override
    public Person process(Person pers) throws Exception {
        Thread.sleep(delay);
        String message="p=" + pers + " processed by thread=" + Thread.currentThread().getName();
        logger.debug(message);
        return pers;
    }
}
```


IV - Tests, Reprise sur erreurs , Monitoring , ...

1. Tests unitaires (pour batch)

NB :

- En version 5 de springBatch il est éventuellement possible de charger en mémoire une configuration complète englobant plusieurs jobs pour ne sélectionner et lancer qu'un seul job au niveau d'un test unitaire. Ceci n'est néanmoins pas une bonne pratique et ce n'est pas possible/prévu en version 4 de SpringBatch.
- Pour qu'un test unitaire de Job puisse bien fonctionner en V4 et V5 , il faut veiller à ne charger uniquement que la configuration d'un seul Job bien précis en codant si besoin une petite classe rassemblant finement toutes les dépendances nécessaires .

1.1. Test unitaire pour job

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.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.boot.test.context.SpringBootTest;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.Configuration;
//....

@Configuration
@EnableAutoConfiguration //springBoot & spring-boot-starter-batch autoConfig (application.properties)
@Import({ AutomaticSpringBootTestJobRepositoryConfig.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());
}
}

```

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.springframework.context.annotation.ImportResource;

@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 {
    // ...
}

```

1.2. Test unitaire pour "step" individuel

NB : Même s'il s'agit d'un test unitaire portant sur un seul "step" , la classe de test doit être configurée pour accéder à la configuration d'un job englobant complet (via la classe habituelle de configuration des dépendances nécessaires).

Ceci permettra à `jobLauncherTestUtils` de partir d'un job de manière à accéder à une de ses étapes (step à lancer/tester).

Exemple :

```
//...

@SpringBatchTest
@SpringBootTest(classes = { FromCsvWithSkipsErrorsToJsonTestConfig.class } )
@ActiveProfiles(profiles = {})
public class TestIndividualStepOfCsvWithSkipsErrorsToJson
    extends AbstractBasicTestJobHelper {

    @Test //unit test of a single/individual Step
    public void testStepCsvWithSkipsErrorsToJson() throws Exception{

        JobExecution jobExecution = jobLauncherTestUtils.launchStep(
            "stepCsvWithSkipsErrorsToJson", initJobParameters());
        //excepted set name = full_name of the step integrated in the job

        Collection<StepExecution> actualStepExecutions = jobExecution.getStepExecutions();
        assertTrue(actualStepExecutions.size()==1);
        StepExecution stepExecution = actualStepExecutions.iterator().next();//first of collection

        //assertions spécifiques à un step d'un job :
        long skipCount=stepExecution.getReadSkipCount();
        logger.debug("*** skipCount="+skipCount );
        assertTrue(skipCount==5); //fichier data/input/inputDataWithErrors.csv with 5 errors

        long writeCount=stepExecution.getWriteCount();
        logger.debug("*** writeCount="+skipCount );
        assertTrue(writeCount>=5); //fichier data/input/inputDataWithErrors.csv with 5 good lines
        //so data/output/outputData.csv must have 5 lines (+header)

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

NB : l'éventuelle classe abstraite **AbstractBasicTestJobHelper** sera présentée sur la page d'après .

1.3. Valideurs/assertions classiques et classe abstraite

Un besoin assez fréquent consiste à vérifier qu'un fichier a bien été généré (avec un bon contenu). On peut pour cela s'appuyer sur un code de ce genre :

```
FileSystemResource expectedResult = new FileSystemResource(expectedFilePath);
FileSystemResource actualResult = new FileSystemResource(actualFilePath);
//AssertFile.assertFileEquals(expectedResult, actualResult); //deprecated since v5
assertThat(actualResult.getFile()).hasSameTextualContentAs(expectedResult.getFile()); //via AssertJ
```

De manière à factoriser efficacement ce code de test , on pourra éventuellement se préparer des classes abstraites réutilisables telles que celles ci :

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 jobInstance 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.assertFileEquals(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{
}

```

2. Reprises sur erreurs

Skip	On saute/skip/ignore certaines erreurs (en fixant une limite et en gardant une trace des lignes en erreurs) . Le Step et le Job est alors considéré comme "COMPLETED" sans erreur grave
Retry	On réessaye plusieurs fois une opération au sein d'un step (avec une éventuelle pause entre deux essais)
Restart	Si le try/retry ne suffit pas on peut relancer le job/batch complet

2.1. Skip some errors

Exemple de fichier (à lire) avec des erreurs :

inputDataWithErrors.csv

```

firstName;lastName;age;active
Jean;Bon;41;true
Alex;Therieur;42;false
LINE-WITH-ERROR-A
LINE-WITH-ERROR-B
LINE-WITH-ERROR-C
Axelle;Aire;33;true
Olie;Condor;44;false
LINE-WITH-ERROR-D
LINE-WITH-ERROR-E
Laurent;Houtan;50;false

```

Configuration XML :

```

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

  <step id="step1_of_fromCsvWithSkipsErrorsToJsonJob" >
    <tasklet>
      <chunk reader="personCsvWithErrorsFileReader" writer="personJsonFileItemWriter"
        processor="simpleUppercasePersonProcessor" commit-interval="1" skip-limit="5" >
        <skippable-exception-classes>
          <!-- <include class="org.springframework.batch.item.file.FlatFileParseException" /> -->
          <include class="org.springframework.batch.item.ItemReaderException" />
        </skippable-exception-classes>
      </chunk>
    </tasklet>
    <listeners>
      <listener ref="mySkippedErrorsListener" />
    </listeners>
  </step>

  <listeners>
    <listener ref="jobCompletionNotificationListener" />
  </listeners>
</job>

```

Configuration Java :

```

return stepBuilder
    .<Person, Person>chunk(5, batchTxManager)
    .listener(new MySkippedErrorsListener())
    .reader(personItemReader)
    .processor(simpleUppercasePersonProcessor)
    .writer(personItemWriter)
    .faultTolerant().skipLimit(5).skip(ItemReaderException.class)
    .listener(new MySkippedErrorsListener())
    .build();

```

//NB: .skipLimit(globalInclusiveMaxLimit) , not consecutive error count but global count

Exemple de "SkipListener" :

```

package tp.mySpringBatch.listener;

import org.springframework.batch.core.SkipListener;
//import org.springframework.batch.item.ItemReaderException;
import tp.mySpringBatch.model.Person;

//NB: org.springframework.batch.item.file.FlatFileParseException hérite de ItemReaderException
//ItemReaderException hérite de RuntimeException

//NB: ce listener est à enregistrer au niveau chunk ou tasklet ou step (mais pas job)
//NB: SkipListener hérite de StepListener
public class MySkippedErrorsListener implements SkipListener<Person,Person>{

    @Override
    public void onSkipInRead(Throwable t) {
        System.err.println("SKIPPED_READ_ERROR:" + t.getMessage());
        SkipListener.super.onSkipInRead(t);
    }

    @Override
    public void onSkipInWrite(Person item, Throwable t) {
        System.err.println("SKIPPED_WRITE_ERROR:" + t.getMessage());
        SkipListener.super.onSkipInWrite(item, t);
    }

    @Override
    public void onSkipInProcess(Person item, Throwable t) {
        System.err.println("SKIPPED_PROCESS_ERROR:" + t.getMessage());
        SkipListener.super.onSkipInProcess(item, t);
    }
}

```

NB : en améliorant ce "SkipListener" on pourrait stocker toutes les lignes à problème quelque part (dans un fichier d'erreurs , de logs , ...).

Autres possibilités :

- Interface **SkipPolicy** avec boolean **shouldSkip**(Throwable throwable, int skipCount) sachant que par défaut la classe d'implémentation **LimitCheckingItemSkipPolicy** se base uniquement sur le paramètre **skipLimit** .
- Annotations **@OnSkipInRead** , **@OnSkipInWrite** , **@OnSkipInProcess** si l'on préfère coder le "skipListener" avec des annotations que via une implémentation d'interface.

Comportement sans "skip" :

"FAILED"

Comportement avec "skip" et "SkipListener" sans dépasser skipLimit :

SUCCESS / "COMPLETED"

et

SKIPPED_READ_ERROR:Parsing error at line: 4 in resource=[URL [file:data/input/csv/inputDataWithErrors.csv]],
input=[LINE-WITH-ERROR-A]

...

SKIPPED_READ_ERROR:Parsing error at line: 10 in resource=[URL [file:data/input/csv/inputDataWithErrors.csv]],
input=[LINE-WITH-ERROR-E]

2.2. Retry when errors in step

Configuration Xml :

```
<step id="step1_of_fromCsvToJsonWithRetryJob" >
  <tasklet>
    <chunk reader="personCsvFileReader" writer="personJsonFileItemWriter"
      processor="uppercasePersonProcessorWithFailuresForRetry"
      commit-interval="1" retry-limit="3" >
      <retryable-exception-classes>
        <include class="tp.mySpringBatch.exception.MyProcessException" />
      </retryable-exception-classes>
    </chunk>
  </tasklet>
</step>
```

Configuration Java (avec peut être un bug) :

```
return stepBuilder
    .<Person, Person>chunk(5, batchTxManager)
    .listener(new MySkippedErrorsListener())
    .reader(personItemReader)
    .processor(uppercasePersonProcessorWithFailuresForRetry)
    .writer(personItemWriter)
    .faultTolerant()
```



```
//backOffPolicy(new ExponentialBackOffPolicy())  
.retryLimit(3).retry(MyProcessException.class) //BUG IN JAVA (not XML) !!!!  
//MyProcessException a non Skippable exception !!!!  
.build();
```

Possibilités sur retry:

- **NoBackOffPolicy** (pas de pause entre 2 retry , par défaut)
ou bien **ExponentialBackOffPolicy** (pause exponentielle entre 2 retry)
- **RetryListener** (facultatif)

NB: depuis la version 2.2 , la fonctionnalité "Retry a été externalisée" (plus dans le coeur de springBatch mais dans la librairie additionnelle "Spring Retry").

Exemple de processeur permettant de tester simplement un "retry" :

```
...  
@Component  
@StepScope  
//@JobScope  
public class UppercasePersonProcessorWithFailuresForRetry implements  
ItemProcessor<Person,Person>{  
    public static int numberOfFailures=0;  
    public static int maxRetry=3;  
  
    @Override  
    public Person process(Person pers) throws Exception {  
        String lastName = pers.getLastName().toUpperCase();  
        Person person=new Person(pers.getFirstName(),lastName,pers.getAge(),pers.getActive());  
  
        numberOfFailures++;  
        if(numberOfFailures==maxRetry)  
            numberOfFailures = 0;  
        if(numberOfFailures>0) {  
            throw new MyProcessException("processExceptionSimulation " +  
                "with numberOfFailures="+numberOfFailures);  
        }else {  
            return person;  
        }  
    }  
}
```

2.3. Restart of job/batch

Lorsqu'une exécution d'un batch n'a pas pu être menée à bien il est quelquefois possible de redémarrer le batch/job complet.

IMPORTANT :

L'exécution d'un job ne pourra redémarrer que si les conditions suivantes sont vérifiées :

- un redémarrage est explicitement demandé via une instruction telle que `jobOperator.restart(mostRecentJobExecution.getId());`
- Le "**status**" (**BatchStatus** , pas **ExitStatus**) d'un **step** est soit **FAILED** ou **STOPPED**

Piège : contrairement au "skip" , le statut d'un step analysé et pris en compte lors d'un restart n'est pas "**ExitStatus**" mais **status** (de type **BatchStatus**) .

NB : le "status" du job entier sera normalement automatiquement fixé à FAILED ou STOPPED si le "status" d'un step est FAILED ou STOPPED .

NB : SpringBatch mémorise des métadonnées au sein du JobRepository .

Ceci permet de reprendre les traitements là où ça s'était arrêté sur erreur .

Depuis la v5 de SpringBatch , le contenu de la colonne **SHORT_CONTEXT** de la table **BATCH_STEP_EXECUTION_CONTEXT** est du **JSON encodé en base64** .

NB : Lorsque l'exécution d'un job est redémarrée , par défaut , seuls les "steps" ayant le status "**FAILED/STOPPED**" seront relancés et les steps réussis (avec status **COMPLETED**) ne le seront pas.

Configuration XML d'un job redémarrable :

```
<job restartable="true">
...
</job>
```

et

```
<tasklet start-limit="3">
....
</tasklet>
```

Si un job comporte plusieurs "step" (un qui réussit , l'autre qui échoue) , on peut éventuellement demander la ré-exécution d'un step réussi via

```
<tasklet allow-start-if-complete="true"> .... </tasklet>
```

Configuration Java pour job redémarrable :

```

@Configuration
public class SimpleRestartableJob extends MyAbstractJobConfig {
    @Bean
    public Job simpleCounterRestartableJob(
        @Qualifier("basicPrintMessageStep") Step step1,
        @Qualifier("simpleCounterStep") Step step2
    ) {
        var name = "simpleCounterRestartableJob";
        var jobBuilder = new JobBuilder(name, jobRepository);
        return jobBuilder.start(step1).next(step2).build();
    }

    ...

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

    @Bean
    @Qualifier("simpleCounterStep")
    public Step simpleCounterStep(
        @Qualifier("simpleIncrementReader") ItemReader<Integer> reader,
        @Qualifier("simpleCounterWriter") ItemWriter<Integer> writer) {
        var name = "simpleCounterStep";
        var stepBuilder = new StepBuilder(name, jobRepository);
        return stepBuilder.<Integer, Integer>chunk(5, batchTxManager)
            .startLimit(3)//all starts (first attemp plus restarts)
            .reader(reader)
            .writer(writer)
            .listener(new MyStoppedForRestartExecutionListener())
            .build();
    }
}

```

```

/* Si plus de 3 démarrages (1 start plus 2 restarts):

```

```

    org.springframework.batch.core.StartLimitExceededException:
    Maximum start limit exceeded for step: simpleCounterStepStartMax: 3 */

```

Persistence d'informations pour une reprise sur erreur :

Pour qu'un job soit redémarrable il faut que ses différentes composantes le soient également (ex : reader et writer en mode "restartable"). Certains (prédéfinis ou pas) le sont , d'autres pas.

Au sein de Spring Batch, l'interface *ItemStream* fourni un moyen de maintenir un état lors d'un redémarrage d'un job .

Exemple : *RestartableCounterIncrementReader.java*

```
...
public class RestartableCounterIncrementReader
    implements ItemReader<Integer> , ItemStream{
    private Logger logger = LoggerFactory.getLogger(RestartableCounterIncrementReader.class);
    private Integer counter=0;

    @Override
    public Integer read() throws Exception, UnexpectedInputException,
        ParseException, NonTransientResourceException {
        counter++;
        logger.debug("#### RestartableCounterIncrementReader.read(): counter="+counter);
        if(counter % 10 == 0)
            return null;
        else
            return counter;
    }

    @Override
    public void open(ExecutionContext executionContext) throws ItemStreamException {
        this.counter = executionContext.getInt("counter", 0);
        logger.debug("##### RestartableCounterIncrementReader.open(): counter="+counter);
    }

    @Override
    public void update(ExecutionContext executionContext) throws ItemStreamException {
        executionContext.putInt("counter", counter);
        logger.debug("##### RestartableCounterIncrementReader.update(): counter="+counter);
    }
}
```

NB : Souvent utile au sein d'une classe de "Reader" customisé , l'interface *ItemStream* comporte 3 méthodes (ayant des implémentations vides par défaut).

Les méthodes "open" et "update" comportent un **paramètre d'entrée** qui correspond à "ExecutionContext" d'un step qui est persisté dans la table **BATCH_STEP_EXECUTION_CONTEXT** du JobRepository .

La méthode .update() est **régulièrement déclenchée** pour le l'on puisse stocker une valeur importante (ici le "counter") dans le "executionContext" qui sera persisté et qui pourra être ré-analysé après un éventuel redémarrage.

La méthode .open() permet de relire l'information stockée au moment d'un redémarrage .

Simulation d'un échec pour test de redémarrage :

```

package tp.mySpringBatch.listener;
import org.springframework.batch.core.BatchStatus;
import org.springframework.batch.core.StepExecution;
import org.springframework.batch.core.StepExecutionListener;

public class MyStoppedForRestartExecutionListener implements StepExecutionListener {

    /* @Override
    public void beforeStep(StepExecution stepExecution) {
        StepExecutionListener.super.beforeStep(stepExecution);
    } */

    @Override
    public ExitStatus afterStep(StepExecution stepExecution) {
        BatchStatus batchStatus = BatchStatus.STOPPED;
        //NO RESTART with default status of stepExecution = BatchStatus.COMPLETED
        //RESTART with status = BatchStatus.FAILED or BatchStatus.STOPPED
        //IMPORTANT: job/step RESTART depends of BatchStatus (but not ExitStatus !!!)
        stepExecution.setStatus(batchStatus);
        return StepExecutionListener.super.afterStep(stepExecution);
    }
}

```

```

stepBuilder.<Integer, Integer>chunk(5, batchTxManager)
                .startLimit(3).reader(reader).writer(writer)
                .listener(new MyStoppedForRestartExecutionListener())
                .build()

```

Code de parcours des éléments du JobRepository d'un Job :

```

public void findAndShowMostRecentJobExecution(String jobName) {
    JobExecution mostRecentJobExecution=null;
    try {
        JobExplorer jobExplorer = applicationContext.getBean(JobExplorer.class);

        List<String> jobNames = jobExplorer.getJobNames();
        System.out.println("jobNames=" + jobNames);
        long nbInstances = jobExplorer.getJobInstanceCount(jobName);
        List<JobInstance> jobInstances = jobExplorer.findJobInstancesByJobName(jobName,
                                                                              0, (int)nbInstances);

        System.out.println("jobInstances=" + jobInstances);

        JobInstance mostRecentJobInstance = jobInstances.get(0);
        System.out.println("mostRecentJobInstance=" + mostRecentJobInstance);

        List<JobExecution> jobExecutions =

```

```

        jobExplorer.getJobExecutions(mostRecentJobInstance);
        System.out.println("jobExecutions=" + jobExecutions);
        mostRecentJobExecution = jobExecutions.get(0);
        System.out.println("mostRecentJobExecution=" + mostRecentJobExecution);
        System.out.println("executionContext of mostRecentJobExecution=" +
            mostRecentJobExecution.getExecutionContext());

        var stepExecutions = mostRecentJobExecution.getStepExecutions();
        for(StepExecution stepExecution : stepExecutions) {
            System.out.println("\t stepExecution with exitStatus="+ stepExecution.getExitStatus()
                +" with status="+ stepExecution.getStatus()+ " and with executionContext="
                + stepExecution.getExecutionContext());
        }

        } catch (Exception e) {
            logger.error(e.getMessage(), e);
        }
        return mostRecentJobExecution;
    }

```

Code de redémarrage d'un Job :

```

public void restartUncompletedJob(String jobName) {
    JobExecution mostRecentJobExecution = this.findAndShowMostRecentJobExecution(jobName);
    if(mostRecentJobExecution!=null)
        this.restartJobExecution(mostRecentJobExecution);
}

public void restartJobExecution(JobExecution jobExecution) {
    try {
        System.out.println("***** restartJobExecution *****");
        JobOperator jobOperator= applicationContext.getBean(JobOperator.class);
        jobOperator.restart(jobExecution.getId());
    } catch (Exception e) {
        logger.error(e.getMessage(), e);
    }
}

```

3. "ItemReaders" , "ItemWriters" personnalisés

3.1. "ItemReader" personnalisé :générer jeux de données

```
public abstract class AbstractPersonGenerator {

    protected long dataSetSize=20; //ou 10000 ou autre
    protected long index=0;

    //most frequent lastNames list
    protected List<String> lastNameList = Arrays.asList("Martin" , "Bernard" , "Thomas" ,
        "Petit" , "Robert" , "Richard" , "Dubois" , "Durand" , ...);

    //most frequent firstNames list
    protected List<String> firstNameList = Arrays.asList("Gabriel" , "Leo" , "Raphael" ,
        "Jade" , "Louise" , "Ambre" , "Emma" , ... );

    //constructeurs , get/set , ...
}
```

```
public class PersonGeneratorReader extends AbstractPersonGenerator
    implements ItemReader<Person>{

    public PersonGeneratorReader() {    super();    }
    public PersonGeneratorReader(long dataSetSize) { super(dataSetSize);    }

    private Person generatePerson() {
        index++;
        int nbFirstNames = firstNameList.size();
        if(index<=dataSetSize) {
            double randomCoeff = Math.random();
            int age = (int)((100 * randomCoeff) % 100);
            String firstName = firstNameList.get(
                (int)(nbFirstNames * randomCoeff) % nbFirstNames);
            return new Person(firstName , lastName ,age , true );
        }else
            return null;
    }

    @Override
    public Person read() throws Exception, UnexpectedInputException,
        ParseException, NonTransientResourceException {
        return generatePerson();
    }
}
```

3.2. "ItemWriter" personnalisé (écritures multiples)

Exemple de "ItemWriter" personnalisé qui va écrire des données dans deux tables d'une base de données (sachant que la seconde écriture doit récupérer la clef primaire auto incrémentée lors de la première écriture) :

```
package tp.mySpringBatch.writer.custom;

import javax.sql.DataSource;
import org.springframework.batch.item.Chunk;
import org.springframework.batch.item.ItemWriter;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;
import tp.mySpringBatch.model.Employee;

public class NewEmployeeDbWriter implements ItemWriter<Employee>{
    private DataSource dataSource;
    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;
    public NewEmployeeDbWriter() {}

    private final String INSERT_PERSON_SQL =
        "INSERT INTO person (first_name, last_name, age, is_active) VALUES (:firstName,:lastName,:age,:active)";
    private final String INSERT_FUNCTIONS_SQL =
        "INSERT INTO functions(id, function,salary) VALUES (:id,:function,:salary)";

    public Employee insertPersonPartOfEmployee(Employee emp) {
        KeyHolder holder = new GeneratedKeyHolder();
        SqlParameterSource parameters = new MapSqlParameterSource()
            .addValue("firstName", emp.getFirstName())
            .addValue("lastName", emp.getLastName())
            .addValue("age", emp.getAge())
            .addValue("active", emp.getActive());
        namedParameterJdbcTemplate.update(INSERT_PERSON_SQL, parameters, holder);
        emp.setId(holder.getKey().longValue()); //store auto_increment pk in instance to return
        return emp;
    }

    public void insertFunctionsPartOfEmployee(Employee emp) {
        SqlParameterSource parameters = new MapSqlParameterSource()
            .addValue("id", emp.getId())
            .addValue("function", emp.getFunction())
            .addValue("salary", emp.getSalary());
        namedParameterJdbcTemplate.update(INSERT_FUNCTIONS_SQL, parameters);
    }

    @Override
    public void write(Chunk<? extends Employee> chunk) throws Exception {
        // write a list of Employee in 2 tables of the database : person and functions
        for(Employee emp : chunk) {
            emp=insertPersonPartOfEmployee(emp);
            insertFunctionsPartOfEmployee(emp);
        }
    }
}
```



```

    }
}

public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
    namedParameterJdbcTemplate=new NamedParameterJdbcTemplate(dataSource);
}

public NewEmployeeDbWriter(DataSource dataSource) {      super();
    this.setDataSource(dataSource);
}
}

```

NB :

public void write(**List**<? extends Employee> items) throws Exception pour **SpringBatch 4**

public void write(**Chunk**<? extends Employee> chunk) throws Exception pour **SpringBatch 5**

4. Lancement de batch

4.1. Via un "scheduler"

Une application java (basée sur le scheduler "quartz") peut régulièrement lancer certains processus de type "appliSpringBatchExcutantJob" tout en effectuant une analyse des résultats (code de retour ou autres)

4.2. Via Spring cloud data flow

Voir paragraphe suivant (avec points communs).

5. Monitoring de job

5.1. Contexte général

Une application SpringBatch a la seule responsabilité de démarrer un job en mode console (sans utilisateur , ni interface graphique) . A part générer quelques fichiers de logs, rien à faire de plus à l'échelle de l'application SpringBatch.

Etant donné que l'application SpringBatch stocke régulièrement des informations dans une base de données (jobRepositoryDB) , on peut envisager une application annexe de type "**springBatchAdmin**" qui va permettre de visualiser l'état d'avancement des instances de job en récupérant ces informations dans la base de données "jobRepositoryDB" .

5.2. Exemples (ancients et récents)

- L'ancien projet "**spring batch admin**" (plus maintenu depuis 2017) a été remplacé par une des parties de *Spring Cloud Data Flow*.
- Certains développeurs se sont développés leurs propres applications d'administration/supervision en s'appuyant quelquefois sur des interfaces graphiques modernes (ex : angular/react/js + api_rest_pour_spring_batch_admin)

5.3. Via Spring Cloud DataFlow

Dans **pom.xml** la dépendance

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-task</artifactId>
    <version>3.1.0</version>
</dependency>
```

Sert à placer **@EnableTask** sur la classe principale (là où il y a **@SpringBootApplication**) ou bien sur une classe de **@Configuration**.

Ceci permet normalement une prise en charge de cette application via "**spring_cloud_dataflow**" .

"**spring_cloud_dataflow**" correspond à un **gros écosystème (basé à fond sur "docker")** qui permet de :

- déployer des applications sur le cloud
- de les contrôler à distance (démarrage immédiat ou différé)
- de les surveiller à distance (vérifier le statut , code de retour ,)

"spring_cloud_dataflow" remplace maintenant l'ancienne application "springBatch-admin" qui n'est plus maintenue depuis de nombreuses années et qui est désormais incompatible avec la nouvelle structure de la base de données (JobRepository) des versions récentes de springBatch .

6. Essentiel sur "Spring Cloud DataFlow"

6.1. Installation de "Spring cloud DataFlow"

1) préparer un machine capable de prendre en charge "docker" et "docker_compose"

- soit une machine "linux"
- soit une machine "windows 11" avec WSL2
- soit une machine virtuelle linux possiblement prise en charge par "virtualBox" et "Vagrant"
- soit une machine "mac" ou autres

installer "java >=17" , "docker" et "docker_compose" (en versions récentes)

2) installer "spring_cloud_DataFlow" via "docker_compose"

a) télécharger les fichiers à prendre en charge par docker compose :

```
wget -O docker-compose.yml https://raw.githubusercontent.com/spring-cloud/spring-cloud-dataflow/main/src/docker-compose/docker-compose.yml;  
wget -O docker-compose-kafka.yml https://raw.githubusercontent.com/spring-cloud/spring-cloud-dataflow/main/src/docker-compose/docker-compose-kafka.yml;  
wget -O docker-compose-mariadb.yml https://raw.githubusercontent.com/spring-cloud/spring-cloud-dataflow/main/src/docker-compose/docker-compose-mariadb.yml;
```

b) docker-compose-up.sh

```
export HOST_MOUNT_PATH=/vagrant/s_c_data_flow/myapps  
export BP_JVM_VERSION=-jdk17  
export DATAFLOW_VERSION=2.11.3  
export SKIPPER_VERSION=2.11.3  
docker-compose -f docker-compose.yml -f docker-compose-kafka.yml  
-f docker-compose-mariadb.yml up --force-recreate
```

sh docker-compose-up.sh

URL par défaut de la console de "spring_cloud_DataFlow" : **http://localhost:9393/dashboard**

3) installer le "shell" de "spring_cloud_DataFlow"

pour télécharger le shell de springCloudDataFlow:

```
wget -O spring-cloud-dataflow-shell-2.11.3.jar  
https://repo.maven.apache.org/maven2/org/springframework/cloud/spring-cloud-dataflow-shell/2.11.3/spring-cloud-dataflow-shell-2.11.3.jar
```

6.2. Prise en charge d'une application "springBatch"

Récupérer le code source d'une application springBatch et déclencher une construction via maven :

```
cd /vagrant/git-repo  
git clone https://github.com/didier-tp/tp-spring-batch  
cd tp-spring-batch  
cd tp/tpSpringBatch  
mvn package -DskipTests  
cd target  
cp tpSpringBatch-0.0.1-SNAPSHOT.jar /vagrant/s_c_data_flow/myapps/tpSpringBatch-0.0.1-SNAPSHOT.jar
```

```
start-scdf-shell.sh
```

```
java -jar spring-cloud-dataflow-shell-2.11.3.jar
```

```
sh start-scdf-shell.sh
```

```
dataflow:>
```

```
app register --type task --bootVersion 3 --name my-app
--uri file:///home/cnb/scdf/tpSpringBatch-0.0.1-SNAPSHOT.jar
```

```
dataflow:> exit
```

Ou bien via console (<http://localhost:9393/dashboard>)

6.3. Démarrage et supervision

<http://localhost:9393/dashboard>

Applications

ADD APPLICATION(S)

GROUP ACTIONS REFRESH

Name	Type	Version	Spring Boot version	URI
my-app	TASK	0.0.1-SNAPSHOT	3.x	file:///home/cnb/s...

Data Flow

Search for keywords...

Applications

Streams

Streams

Runtime

Tasks / Jobs

Tasks

Task executions

Job executions

Tasks

CREATE TASK

GROUP ACTIONS REFRESH

Name

Create a task

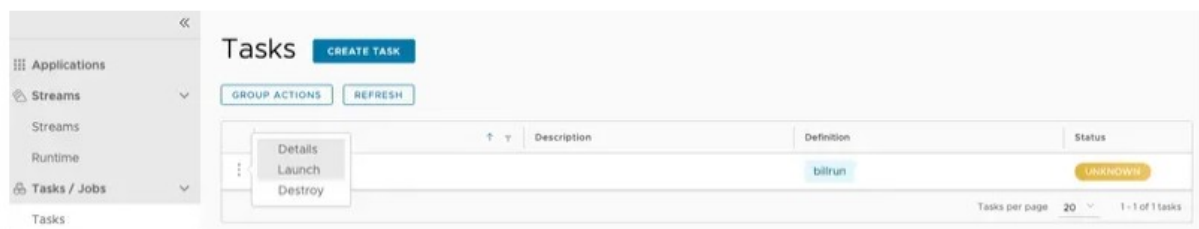
... (à peaufiner) , source : <https://github.com/didier-mycontrib/msa-vagrant> , va-sta-msa-java

Documentation officielle de référence :

<https://dataflow.spring.io/docs/batch-developer-guides/batch/data-flow-spring-batch/>

Launching the Task

The following image shows the Tasks view, from which you can launch a task:



Applications
Streams
Streams
Runtime
Tasks / Jobs
Tasks

Task executions

GROUP ACTIONS REFRESH

Execution ID	Task name	Duration	Start Date	End Date	Exit Code
235	billrun	00:00:00.000	2020-11-17 14:41:15,000[-5]	2020-11-17 14:41:15,000[-5]	0

Task executions per page 20 1 - 1 of 1 task executions

Applications
Streams
Streams
Runtime
Tasks / Jobs
Tasks

Jobs

REFRESH

Execution Id	Name	Task Id	Instance Id	Job Start Time	Step Execution Count	Status
79	BillJob	235	77	2020-11-17 14:41:15,000[-5]	1	COMPLETED

Job executions per page 20 1 - 1 of 1 job executions

Applications
Streams
Streams
Runtime
Tasks / Jobs
Tasks
Tasks executions
Jobs executions
Manage
Audit Records
Import / Export

Job execution BillJob (79)

RESTART THE JOB STOP THE JOB

INFORMATION

Id	79
Job Name	BillJob
Job Instance	77
Task Execution Id	235
Job Parameters	
Start Time	2020-11-17 14:41:15,000[-5]
End Time	2020-11-17 14:41:15,000[-5]
Duration	00:00:00.000
Status	COMPLETED
Exit Code	COMPLETED
Exit Message	
Step Execution Count	1

TASK

Name	billrun
Definition	billrun
Status	COMPLETE
Applications	billrun RELBUN

VIEW TASK RELAUNCH TASK

TASK EXECUTION

VIEW TASK EXECUTION VIEW LOG

STEPS

Step Id	Step Name	Reads	Writes	Commits	Rollbacks	Duration	Status
184	BillProcessing	0	0	1	0	00:00:00.000	COMPLETED

7. Aspects divers et avancés

7.1. logging (pour batch)

Utilisation conjointe de "listener" et de "slf4j" pour générer des fichiers de logs à chaque exécution d'un job .

7.2. Principaux patterns (pour batch)

<https://docs.spring.io/spring-batch/reference/common-patterns.html>

<https://github.com/desprez/springbatch-patterns>

Principal "pattern" :

custom_reader **delegating** .read() to subReader(s) created and open in @BeforeStep method and closed in @AfterStep method .

Idem pour "custom_writer" delegating to subWriter(s) .

ANNEXES

V - Annexe – TP Spring Batch

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 , maven , SpringBoot 3.x (stable) .

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

Dependencies : SpringBatch et h2

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;}
}

```

Cr  er 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");
    }
}

```

Cr  er 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(
        @Qualifier("simplePrintMessageStep") 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 de 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
http://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.configAutomaticSpringBootApplicationRepositoryConfig;

@Configuration
@EnableAutoConfiguration //springBoot & spring-boot-starter-batch autoConfig (application.properties)
@Import({ AutomaticSpringBootApplicationRepositoryConfig.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 pouvoir 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.configAutomaticSpringBootApplicationRepositoryConfig;

@Configuration
@EnableAutoConfiguration //springBoot & spring-boot-starter-batch autoConfig (application.properties)
@Import({ AutomaticSpringBootApplicationRepositoryConfig.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 jobInstance 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.assertFileEquals(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{
}

```

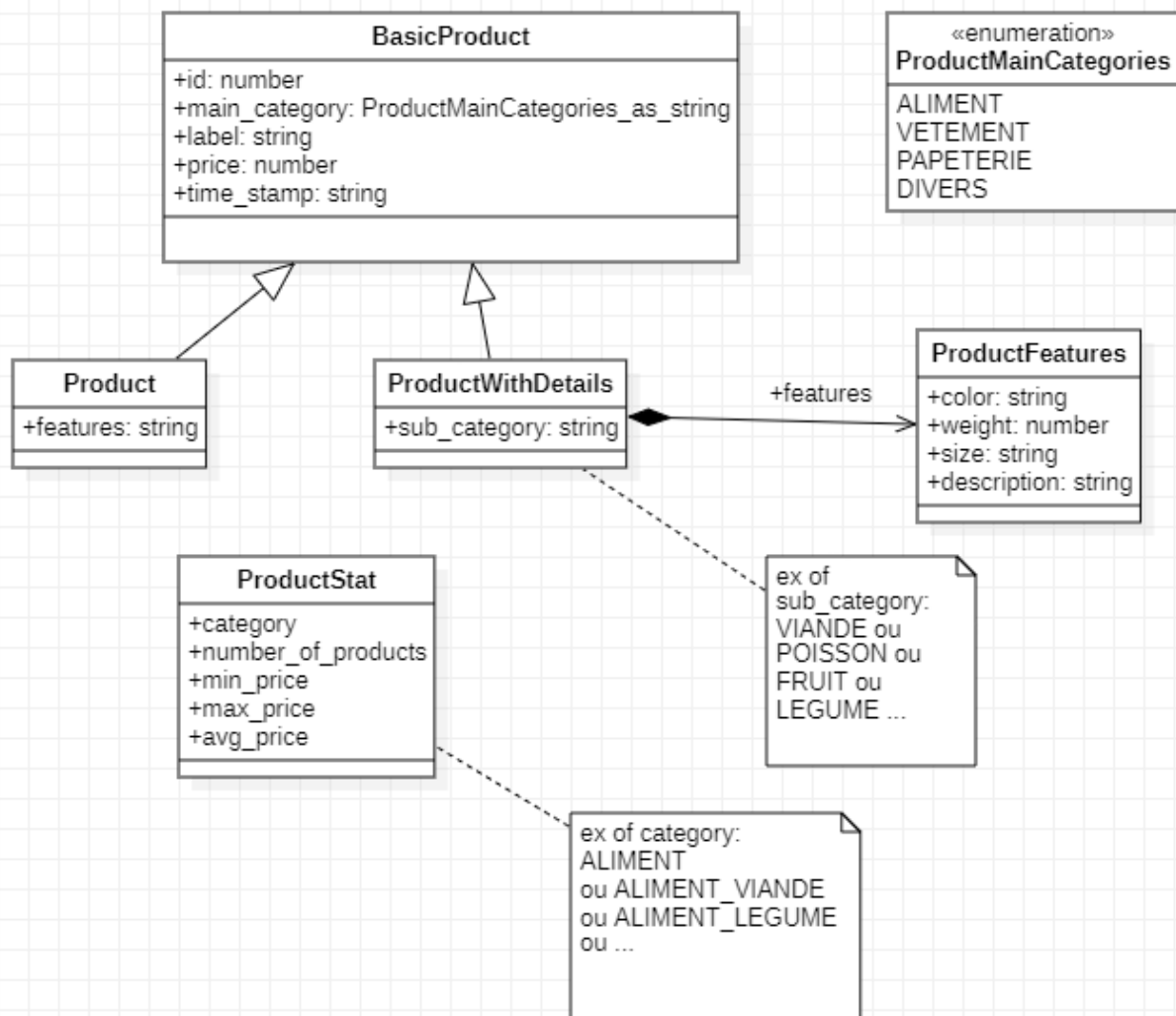
```

...
public class TestXmlHelloWorldJob extends AbstractBasicActiveTestJob{
}

```

Dans la suite des TP's 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/ouput/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 a 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 :

```
@XmlRootElement(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 :

inputFilePath	"data/input/csv/products.csv"
outputFilePath	"data/output/xml/products.xml"
enableUpperCase	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 outputFilePath
) {

//logger.info("in @Bean productXmlFileItemWriter() , outputFilePath="+outputFilePath);
WritableResource outputXmlResource = new FileSystemResource(outputFilePath);
...
}
```

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="file:#{jobParameters['outputFilePath']}" />
  ...
</bean>
```

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 outputFilePath=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\eclipse\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
// (categorie 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 <code>RowMapper<ProductStat></code>
MyDbProductStatReaderConfig	Déclenchant cette requête SQL : <pre>SELECT main_category as category , count(*) as nb_prod , min(price) as min_price , max(price) as max_price , avg(price) as avg_price FROM PRODUCT_WITH_DETAILS GROUP BY main_category</pre>
MyCsvFileProductStatWriterConfig	Générant un fichier productStats.csv

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.