Le traitement par lot, communément appelé Batch dans le jargon informatique, est une problématique très répandue et quasiment incontournable au sein des entreprises et industries qui manipulent d'énormes masses de données. Dans cet article, nous allons vous présenter au travers d'un exemple d'application, la technologie **Spring Batch/Spring Boot** permettant de répondre à ce type de besoin. Cet article ne se veut pas et ne peut pas être exhaustif, car il s'agit d'une technologie immensément vaste. Mais les différents concepts présentés dans ce cours sont suffisants pour son appropriation.



## L'auteur

Georges KEMAYO

## L'article

Publié le 25 mars 2020

TOUT PUBLIC

Version PDF Version hors-ligne

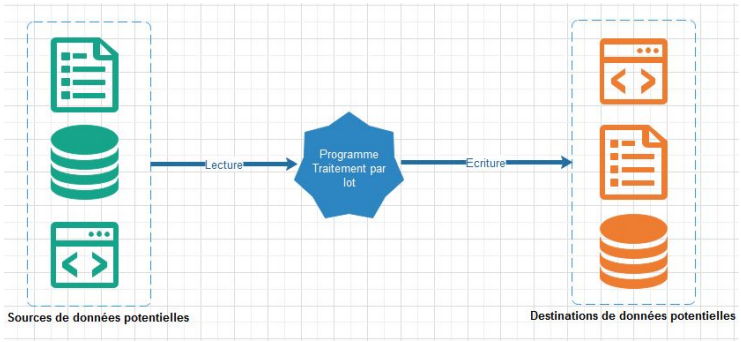
ePub, Azw et Mobi

## Liens sociaux



## I. Qu'est ce qu'un Batch ?▲

Un Batch est une application informatique dont la vocation est d'effectuer du traitement par lot sur une masse de données. En général ce type de programme s'exécute de façon autonome, peut potentiellement consommer beaucoup de ressources machine (CPU, mémoire, etc.) et peut avoir un temps d'exécution long (plusieurs minutes ou heures) lorsque le volume de données à traiter est consistant. Il est en général conçu pour se déclencher et s'exécuter de façon autonome, automatique et régulière/périodique, mais ce n'est pas obligatoire, car la gestion d'un batch peut tout à fait être manuelle. Un Batch a pour objectif de lire des données provenant de diverses sources homogènes ou hétérogènes (fichiers, bases de données, web, etc.), puis d'effectuer des traitements/transformations sur ces données lues afin d'en stocker le résultat dans un ou plusieurs conteneurs de destination (fichiers, bases de données, queue, etc.) dans le but de les exploiter. Nous pouvons résumer cette définition au travers de l'image ci-dessous :

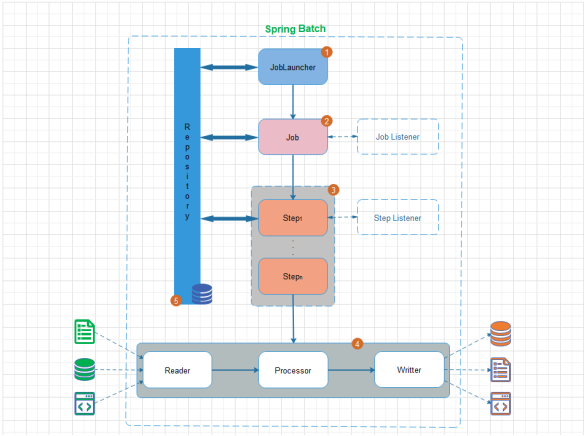



## II. Qu'est-ce que Spring Batch ? Et quel rôle joue Spring Boot ?▲

Spring Batch est un puissant framework Java basé sur Spring et qui permet de construire des programmes de traitement par lot. Si nous reprenons la figure de la section I, nous devinons donc que Spring Batch se positionne au centre entre les deux blocs rectangulaires au niveau de la roue dentée. Plus concrètement, il s'agit d'un framework qui fournit un ensemble de composants et de fonctionnalités réutilisables et personnalisables permettant de donner au programme de traitement par lot tous les éléments nécessaires à :

- la définition d'un cadre structuré pour implémenter les règles métier ;
- son démarrage et la gestion de son ordonnancement ;
- son monitoring et la réalisation de statistiques sur les données et tâches manipulées en son sein ;
- l'écoute et la capturation des évènements qui surviennent avant, pendant et après son exécution ;
- l'amélioration de la performance des traitements à travers des techniques de partitionnement et d'optimisation ;
- etc.

Nous pouvons représenter de façon macroscopique Spring Batch par la figure suivante :




Ce schéma illustre une vue architecturale générale et non détaillée du framework Spring Batch. Il présente plusieurs composants qui doivent être définis et configurés pour suivre le workflow dont le sens suit la numérotation qui accompagne chacun d'eux sur ce schéma.

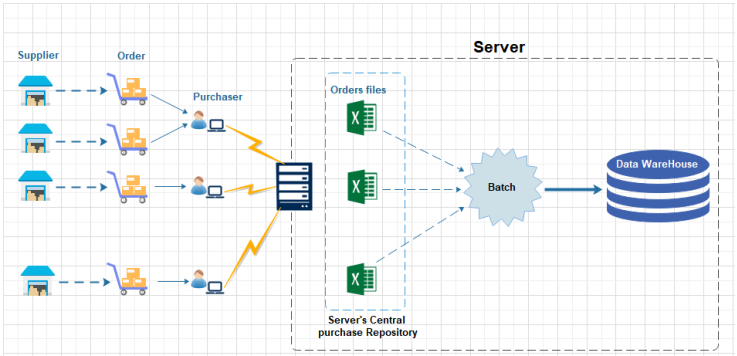
- Le **JobLauncher :** il s'agit du composant chargé de lancer/démarrer le programme de traitement par lot (batch). Il peut être configuré pour s'autodéclencher ou pour être déclenché par un évènement extérieur (lancement manuel). Dans le worflow Spring Batch, le JobLauncher est chargé d'exécuter un Job (tâche).
- Le **Job :** il s'agit du composant qui représente la tâche à qui on délègue la responsabilité du besoin métier traité dans le programme. Il est chargé de lancer de façon séquentielle une ou plusieurs Step.
- La **Step :** c'est le composant qui enveloppe le cœur même du besoin métier à traiter. Il est chargé de définir trois sous-composants structurés comme suit :
  (1) Le **Reader :** c'est le composant chargé de lire les données d'entrées à traiter. Elles peuvent provenir de diverses sources (bases de données, fichiers plats (csv, xml, xls, etc.), queue) ;
  (2) Le **Processor :** c'est le composant responsable de la transformation des données lues. C'est en son sein que toutes les règles de gestion sont implémentées ;
  (3) Le **Writer :** ce composant sauvegarde les données transformées par le processor dans un ou plusieurs conteneurs désirés (bases de données, fichiers plats (csv, xml, xls, etc.), cloud).
- Le **Repository :** c'est le composant chargé d'enregistrer les statistiques issues du monitoring sur le JobLauncher, le Job et la (ou les) Step à chaque exécution. Il offre deux techniques possibles pour stocker ces statistiques : le passage par une base de données ou le passage par une Map. Lorsque le stockage des statistiques est fait dans une base de données, et donc persisté de façon durable, cela permet le suivi continuel du Batch dans le temps à l'effet d'analyser les éventuels problèmes en cas d'échec. A contrario lorsque c'est dans une Map, les statistiques persistées seront perdues à la terminaison de chaque instance d'exécution du Batch. Dans tous les cas, il faut configurer l'un ou l'autre obligatoirement.
- Les **Job et Step Listeners :** ce sont chacun des composants facultatifs qui offrent un cadre pour l'implémentation des services de monitoring personnalisés. Ils sont très utiles pour journaliser (loguer) les états du Job/Step avant et après leur exécution.

Comme vous pouvez déjà vous en apercevoir au travers du schéma et des explications ci-dessus, Spring Batch est un framework complexe et pas très simple à appréhender. Il y a beaucoup de configurations à faire en plus de l'implémentation des règles de gestion constituant le cœur même du besoin métier à résoudre.
C'est à ce niveau que Spring Boot intervient et va nous aider à simplifier la configuration de Spring Batch afin d'améliorer la productivité. Dans l'article suivant, nous avions déjà expliqué et listé les avantages de Spring Boot. Ceux-ci s'appliquent entièrement également pour le cas Spring Batch et constituent la raison fondamentale d'utiliser de nos jours Spring Batch à travers Spring Bootlorsque l'environnement de travail nous en donne la possibilité.

## III. Exemple d'application▲

Pour illustrer comment utiliser Spring Batch dans un programme de traitement par lot, nous allons définir ci-dessous un exemple de sujet qui va nous amener à atteindre notre objectif.

**Sujet :** notre exemple concerne la conception et le développement d'un Batch qui traitera les fichiers internes de commandes d'une société commerciale qui revend du matériel multimédia à grande échelle. L'objectif premier du Batch sera de lire continuellement les données de commandes contenues dans des fichiers csv que dépose, dans un répertoire commun, chaque responsable d'achat lorsqu'il reçoit ses commandes d'un ou plusieurs fournisseurs de la société. Ensuite, le Batch devra les traiter afin de les agréger pour ensuite les stocker dans un entrepôt de données. Nous pouvons résumer le processus d'achats par le workflow illustré par la figure ci-dessous :




Notre Batch jouera alors le rôle de ce que l'on qualifie dans le domaine de la Business Intelligence(BI), un **ETL** (Extract-Transform-Load). Les données agrégées et stockées dans l'entrepôt de données permettront d'aider à la prise de décision par le(s) responsable(s) de la société sur les orientations possibles de politiques d'achats.

Nous supposerons que chacun des fichiers csv déposés par un agent commercial d'achat est structuré de la façon suivante, où la première ligne représente l'entête composé de l'intitulé de chaque colonne. Les valeurs de chaque colonne sont séparées par des points-virgules :

```
PRDT_NAME;PRDT_EAN_CODE;PRDT_TYPE;PRDT_QTE;PRDT_AMOUNT;SUPPLIER_NAME;SUPPLIER_ADDRESS;PURCHASER_FIRST_NAME;PURCHASER_LAST_NAME;PURCHASER_EMAIL;TRANSACTION_DATE
ZenPhone5;EAN2918098;Phone;2;300.0;ASUS;7 Place Vendome Paris France;Eric;Dupont;ericdupont@yopmail.com;2019-12-18T17:09:42.411
USBZEN;EAN10000;USB KEY;8;15,99;Cdiscount;27 Boulevard des Nations Nantes France;Eric;Dupont;ericdupont@yopmail.com;2019-12-15
CameraZEN;EAN5000;Camera;12;160,57;Cdiscount;27 Boulevard des Nations Nantes France;Samuel;Marimont;samuelmarimont@yopmail.com;2019-12-15
```

e besoin métier voulu est de lire l'ensemble des fichiers déposés à un instant **t** sur le répertoire central du serveur et d'historiser les données lues dans un entrepôt de données de manière à ce que :

1. Pour chaque ligne lue, on puisse directement avoir accès à la somme totale des dépenses effectuées pour un article (somme totale = quantité*prix_unitaire) :
2. Pour chaque ligne lue, on n'effectue pas de doublon en écriture dans la base de données, des données concernant le fournisseur (Supplier), l'agent d'achat (Purchaser) et le produit de même type contenu dans les commandes (Order).

## IV. Infrastructure et Modèle de données▲

## IV-A. Infrastructure▲

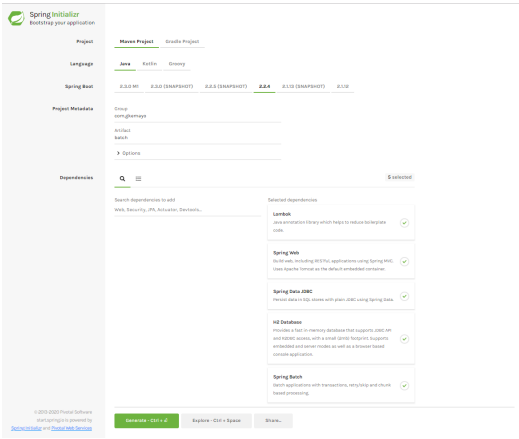
Nous vous l'avons dit précédemment, nous construirons notre application à l'aide de Spring Bootauquel nous allons injecter le framework Spring Batch. Notre application sera packagée en **jar**. Pour ceux qui s'initient encore à la création d'un projet Spring Boot, vous pouvez suivre le tutoriel suivant où nous nous étions déjà appesantis sur la description du processus.

L'architecture de notre application va naturellement suivre celle illustrée à la figure section II déjà tracée par Spring Batch. La seule petite différence dans cet article est que nous ne créerons pas explicitement le composant JobLauncher. En effet, lorsque nous utilisons Spring Batch avec Spring Boot, il crée pour nous par défaut le composant JobLauncher pour permettre le lancement automatique du Batch dès qu'on exécute le jar de l'application (cf. java -jar xxx.jar).

Nous pouvons bien entendu désactiver ce comportement par défaut pour créer notre propre JobLauncher.
Propriété spring.batch.initializer.enabled=false à ajouter dans l'application.properties.

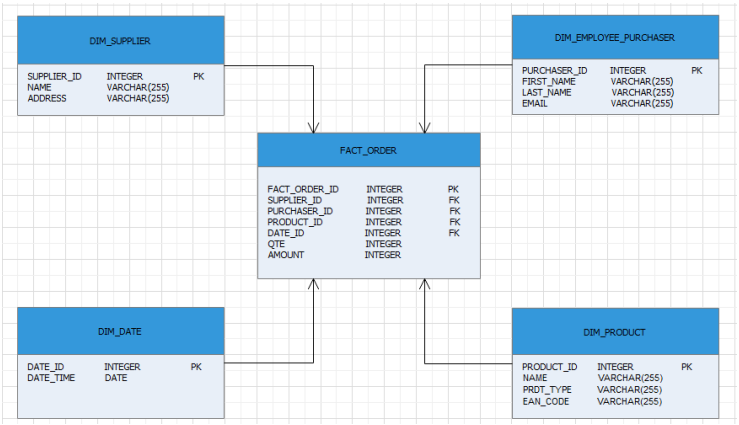
En somme, l'infrastructure de développement de notre application est la suivante :

- **Eclipse SimRel 2018-12 :** l'IDE de développement ;
- **Spring Boot 2.2.4 :** socle de construction du projet ;
- **Spring web 5.2.3 :** pour injecter le tomcat embarqué nécessaire au fonctionnement standalone de l'application ;
- **Java 8 :** langage applicatif utilisé ;
- **Maven 3.2 :** pour la gestion des dépendances ;
- **Lombok 1.18.10 :** pour la gestion des accesseurs des objets Java utilisés dans l'application ;
- **Spring Batch 4.2.0 :** pour permettre d'accéder aux composants permettant de construire notre Batch ;
- **Spring Data JDBC 1.1.1 :** pour la gestion de la persistance des données dans l'entrepôt de données ;
- **H2 Database 1.4.200 :** l'entrepôt de données de l'application.



## IV-B. Modèle de données▲

D'après l'exposé du besoin métier décrit à la section III, nous proposons le modèle physique de données ci-dessous, où nous avons choisi d'utiliser les termes anglo-saxons pour qualifier nos entités.

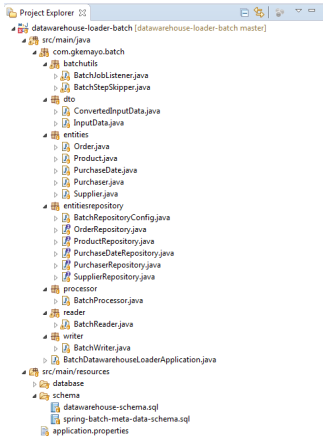



Il s'agit d'un modèle en étoile dans lequel la table FACT_ORDER (table des Commandes) correspond à ce qu'on appelle la table des Faits et toutes les tables qui gravitent autour correspondent aux tables de Dimensions. Dans le cas d'espèce, DIM_PRODUCT (table qui recense sans doublon les produits), DIM_EMPLOYEE_PURCHASER (table qui recense sans doublon les agents d'achats), DIM_SUPPLIER (table qui recense sans doublon les fournisseurs), DIM_DATE(table qui recense sans doublon les dates d'achats) correspondent aux tables de Dimensions. Nous vous invitons à lire l'article suivant pour plus de détails sur la modélisation d'entrepôt de données.

## V. Mise en oeuvre de l'application▲

## V-A. Structure du projet et pom.xml▲

L'application baptisée **datawarehouse-loader-batch** que nous allons construire aura la structuration finale suivante sur Eclipse :




Le pom.xml résultant est le suivant :

Sélectionnez
1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
24.
25.
26.
27.
28.
29.
30.
31.
32.
33.
34.
35.
36.
37.
38.
39.
40.
41.
42.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>2.2.4.RELEASE</version>
		<relativePath/> <!-- lookup parent from repository -->
	</parent>
	<groupId>com.gkemayo</groupId>
	<artifactId>datawarehouse-loader-batch</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>datawarehouse-loader-batch</name>
	<description>Spring Batch project for extracting data on files resources and load them on a DataWarehouse</description>

	<properties>
		<java.version>1.8</java.version>
	</properties>

	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-batch</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-data-jdbc</artifactId>
		</dependency>

		<dependency>

			<groupId>com.h2database</groupId>
			<artifactId>h2</artifactId>
```

```
                <scope>runtime</scope>
            </dependency>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-configuration-processor</artifactId>
                <optional>true</optional>
            </dependency>
            <dependency>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
                <optional>true</optional>
            </dependency>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-test</artifactId>
                <scope>test</scope>
                <exclusions>
                    <exclusion>
                        <groupId>org.junit.vintage</groupId>
                        <artifactId>junit-vintage-engine</artifactId>
                    </exclusion>
                </exclusions>
            </dependency>
            <dependency>
                <groupId>org.springframework.batch</groupId>
                <artifactId>spring-batch-test</artifactId>
                <scope>test</scope>
            </dependency>
        </dependencies>

        <build>
            <plugins>
                <plugin>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-maven-plugin</artifactId>
                </plugin>
            </plugins>
        </build>
</project>
```

- le starter spring-boot-starter-web injectera toutes les dépendances liées à Spring web ;
- le starter spring-boot-starter-batch injectera toutes les dépendances liées à Spring Batch ;
- le starter spring-boot-starter-data-jdbc injectera toutes les dépendances liées à Spring Data JDBC ;
- le starter spring-boot-starter-test injectera toutes les dépendances pour faire des tests unitaires et d'intégration ;
- la dépendance spring-batch-test met à disposition des composants permettant de faciliter le test d'un projet Spring Batch.

## V-B. L'application.properties▲

Sélectionnez

```
1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
############# DataSource Config #################
spring.datasource.name=datawarehouse-db
spring.datasource.username=sa
```

```
spring.datasource.password=
spring.datasource.url= jdbc:h2:file:./src/main/resources/database/datawarehouse-db
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.sql-script-encoding= UTF-8

spring.datasource.schema = classpath:schema/datawarehouse-schema.sql

spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
#http://localhost:8080/h2-console/

logging.level.root=info
logging.file.max-size=10MB
logging.file=./logs/batch-datawarehouse-loader.log

path.to.the.work.dir=../../datawarehouse-reports-repository
```

**Explications :**

- **spring.datasource.* :** ces paramètres permettent de configurer la datasource permettant d'accéder à la base de données H2 de l'application. Nous choisissons pour cet exemple l'application une persistance durable de type file. Dans une application d'envergure, il faut bien entendu une base de données d'entreprise telle que Oracle, Postgresql, SQL Server, MySQL, etc.
- **spring.datasource.schema :** permet au démarrage de l'application Spring Boot de créer le schéma (tables, index, etc.) de notre base de données (FACT_ORDER, DIM_SUPPLIER, DIM_PRODUCT, DIM_DATE, DIM_PURCHASER). Par défaut, Spring Boot crée le schéma des métadonnées Spring Batch (c'est-à-dire, les tables de statistiques de Spring Batch) lorsque le droit d'écriture sur la base de données est autorisé. Dans le cas contraire, il faut le jouer manuellement. Nous vous avons donc mis le script SQL concerné dans le fichier spring-batch-meta-data-schema.sql du projet.
- **spring.h2.console.* :** permet la visualisation web de la base de données H2 sur l'URL http://localhost:8080/h2-console/.
- **logging.* :** configure les paramètres permettant la journalisation (log) des événements qui surviennent dans l'application.
- **path.to.the.work.dir :** paramètre personnel qui indique l'endroit où trouver le répertoire central de dépôt des fichiers de commandes.

Pour rappel, l'ensemble des paramètres standards Spring Boot peuvent être retrouvés sur la page suivante.

## V-C. La classe de configuration globale▲

Sélectionnez
1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
24.
25.
26.
27.
28.
29.
30.
31.
32.
33.

```
34.
35.
36.
37.
38.
39.
40.
41.
42.
43.
44.
45.
46.
47.
48.
49.
50.
51.
52.
53.
54.
55.
56.
57.
58.
59.
60.
61.
62.
63.
64.
65.
66.
67.
68.
69.
70.
71.
72.
73.
74.
75.
76.
77.
78.
79.
80.
81.
82.
83.
84.
85.
86.
87.
88.
89.
90.
91.
92.
93.
94.
95.
96.
97.

98.
99.
```

```
package com.gkemayo.batch;

import javax.sql.DataSource;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.core.launch.support.RunIdIncrementer;
import org.springframework.batch.core.repository.JobRepository;
import org.springframework.batch.core.repository.support.JobRepositoryFactoryBean;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.transaction.PlatformTransactionManager;

import com.gkemayo.batch.batchutils.BatchJobListener;
import com.gkemayo.batch.batchutils.BatchStepSkipper;
import com.gkemayo.batch.dto.ConvertedInputData;
import com.gkemayo.batch.dto.InputData;
import com.gkemayo.batch.processor.BatchProcessor;
import com.gkemayo.batch.reader.BatchReader;
import com.gkemayo.batch.writer.BatchWriter;

@SpringBootApplication
@EnableBatchProcessing
public class BatchDatawarehouseLoaderApplication {

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

	@Value("${path.to.the.work.dir}")
	private String workDirPath ;

	@Autowired
	private DataSource dataSource;

	@Autowired
	PlatformTransactionManager transactionManager;

	@Bean
	public JobRepository jobRepositoryObj() throws Exception {
		JobRepositoryFactoryBean jobRepoFactory = new JobRepositoryFactoryBean();
		jobRepoFactory.setTransactionManager(transactionManager);
		jobRepoFactory.setDataSource(dataSource);
		return jobRepoFactory.getObject();
	}

	@Autowired
	JobBuilderFactory jobBuilderFactory;

	@Autowired
	StepBuilderFactory stepBuilderFactory;

	@Bean
	public BatchReader batchReader() {
		return new BatchReader(workDirPath);
	}

	@Bean
	public BatchProcessor batchProcessor() {
		return new BatchProcessor();
	}

	@Bean
	public BatchWriter batchWriter() {
		return new BatchWriter();
	}


	@Bean
```

```
	public BatchJobListener batchJobListener() {
		return new BatchJobListener();
	}

	@Bean
	public BatchStepSkipper batchStepSkipper() {
		return new BatchStepSkipper();
	}

	@Bean
	public Step batchStep() {
		return stepBuilderFactory.get("stepDatawarehouseLoader").transactionManager(transactionManager)
				.<InputData, ConvertedInputData>chunk(1).reader(batchReader()).processor(batchProcessor())
				.writer(batchWriter()).faultTolerant().skipPolicy(batchStepSkipper()).build();
	}

	@Bean
	public Job jobStep() throws Exception {
		return jobBuilderFactory.get("jobDatawarehouseLoader").repository(jobRepositoryObj()).incrementer(new RunIdIncrementer()).listener(batchJobListener())
				.flow(batchStep()).end().build();
	}

}
```

**Explications**

**- Les annotations**

- **@SpringBootApplication :** annotation Spring Boot qui encapsule les trois annotations suivantes : @Configuration, @EnableAutoConfiguration, @ComponentScan.
- **@EnableBatchProcessing :** annotation qui déclare une classe comme une qui va permettre de créer et configurer des composants Spring Batch.

**- Les composants Spring Batch**

- **JobRepository :** correspond au composant Repository de l'architecture Spring Batch numéroté 5 sur la figure de la section II.
- **JobBuilderFactory :** classe native Spring Batch qui offre des fonctionnalités permettant de créer et d'enrichir un **Job** de tous les éléments dont il a besoin.
- **StepBuilderFactory :** classe native Spring Batch qui offre des fonctionnalités aidant à créer et enrichir une **Step** de tous les éléments dont elle a besoin.
- **BatchReader :** classe qui effectuera la lecture des données dans les fichiers csv entrés.
- **BatchProcessor :** classe qui effectuera les traitements nécessaires sur les données lues.
- **BatchWriter :** classe qui va sauvegarder les données traitées dans l'entrepôt de données.
- **BatchJobListener :** classe qui permettra de journaliser les évènements pendant l'exécution du Job.
- **BatchStepSkipper :** classe qui va permettre la gestion de la tolérance aux fautes. En d'autres termes, cette classe va permettre au Batch de tolérer certaines lignes non conformes lues par le Reader en les sautant tout simplement, pour ne pas pénaliser toutes les autres lignes qui pourraient être conformes. En général un seuil est paramétré et c'est à l'atteinte de ce seuil que la tolérance aux fautes n'est plus acceptée et le Batch s'arrêtera donc.
- **Step :** classe native Spring Batch à laquelle on enrichira le Reader, le Processor et le Writerdu Batch. De plus, on va lui indiquer le Pojo dans lequel seront enregistrées les données lues par le Reader depuis les fichiers d'entrées (ici, InputData) et sous quel format le Processoraprès les avoir transformées les délivrera au Writer (ici, ConvertedInputData). L'argument chunk définit le nombre de données lues et traitées à la fois par le Reader. À savoir qu'un Reader Spring Batch partitionne les données d'entrées et les lit par lot (défini donc par le chunk). Enfin, nous valorisons le champ skipPolicy avec le bean qui va se charger de gérer la tolérance aux fautes (en l'occurrence, BatchStepSkipper).
- **Job :** classe native Spring Batch dans laquelle on enrichira notre Step précédente, le listeneret le jobRepository.

Vous avez pu remarquer que l'ensemble de ces composants/classes Spring Batch sont targués par les annotations @Autowired et @Bean. Ce qui signifie qu'ils sont injectés dans l'application comme des beans Spring, et donc sont des singletons.

## V-D. La classe de configuration Spring Data JDBC▲

Spring Data JDBC est un framework permettant la gestion de la persistance des données dans une application. Il s'agit d'une couche d'abstraction placée au-dessus de l'API (Application Programming Interface) JDBC que vous connaissez tous. Spring Data JDBC est une version allégée de Spring Data JPA comme vous pouvez le voir dans sa documentation officielle suivante.

Ce qui nous a motivés à choisir Spring Data JDBC dans cette application est que nous n'avions pas besoin d'effectuer des opérations complexes sur les entités telles que le Lazy loading, les mapping ORM complexes, les transactions. En effet, une contrainte de notre entrepôt de données, à savoir ne pas enregistrer les données en doublon, nous oblige à commiter après chaque opération de sauvegarde (cf. chunk(1))...

Sélectionnez
1.
2.
3.
4.
5.

```
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
24.
25.
```

```
package com.gkemayo.batch.entitiesrepository;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jdbc.repository.config.AbstractJdbcConfiguration;
import org.springframework.data.jdbc.repository.config.EnableJdbcRepositories;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcOperations;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;

@Configuration
@EnableJdbcRepositories
public class BatchRepositoryConfig extends AbstractJdbcConfiguration {

        @Autowired
        private DataSource dataSource;

        @Bean
    NamedParameterJdbcOperations operations() {
        return new NamedParameterJdbcTemplate(dataSource);
    }

}
```

**Explications**

Pour utiliser Spring Data JDBC dans une application, il faut lui créer une classe de configuration (qui sera targuée **@Configuration**) Spring à laquelle on ajoutera l'annotation **@EnableJdbcRepositories**. Cette classe devra également étendre la classe **AbstractJdbcConfiguration**. Enfin, dans la classe de configuration, il faudra définir un Bean **NamedParameterJdbcOperations** qui indiquera à Spring Data JDBC où trouver la datasource de l'application.

## V-E. La Classe Reader▲

Sélectionnez

```
1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.

12.
13.
```

```
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
24.
25.
26.
27.
28.
29.
30.
31.
32.
33.
34.
35.
36.
37.
38.
39.
40.
41.
42.
43.
44.
45.
46.
47.
48.
49.
50.
51.
52.
53.
54.
55.
56.
57.
58.
59.
60.
61.
62.
63.
64.
65.
66.
67.
68.
69.
70.
71.
72.
73.
74.
75.
76.
77.
78.
79.
80.
```

```
package com.gkemayo.batch.reader;

import java.io.File;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.MultiResourceItemReader;
import org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper;
import org.springframework.batch.item.file.mapping.DefaultLineMapper;
import org.springframework.batch.item.file.mapping.FieldSetMapper;
import org.springframework.batch.item.file.transform.DelimitedLineTokenizer;
import org.springframework.batch.item.file.transform.FieldSet;
import org.springframework.core.io.FileSystemResource;
import org.springframework.validation.BindException;

import com.gkemayo.batch.dto.InputData;

import lombok.extern.slf4j.Slf4j;

/**
 * Spring batch reader : read all the input data files
 *
 * @author Georges Kemayo
 *
 */
@Slf4j
public class BatchReader extends MultiResourceItemReader<InputData>{

	Logger log = LoggerFactory.getLogger(BatchReader.class);

	public BatchReader(String workDirPath) {
		log.info("Batch reader starting to read input data in repository : " + workDirPath);
		this.setResources(getInputResources(workDirPath));
		this.setDelegate(readOneFile());
	}
```

```
	/**
	 * Get all the resources (files) to be readen by this spring batch reader.
	 *
	 * @param workDirPath
	 * @return
	 */
	private FileSystemResource[] getInputResources(String workDirPath) {

		File inputDir = new File(workDirPath);
		File[] files = inputDir.listFiles();
		List<File> filesList = Arrays.asList(files);

		List<FileSystemResource> inputResources = filesList.stream().filter(file -> file != null && file.isFile())
                                                                    .peek(file -> log.info("Reading file : " + file.getAbsolutePath()))
                                                                      .map(file -> new FileSystemResource(file))
                                                                       .collect(Collectors.toList());

		return inputResources.toArray(new FileSystemResource[inputResources.size()]);
	}

	/**
	 * Set and return a FlatFileItemReader to read one resource.
	 *
	 * @return
	 */
	private FlatFileItemReader<InputData> readOneFile() {

		FlatFileItemReader<InputData> resourceReader = new FlatFileItemReader<InputData>();

		//skip the first line which is the file header
		resourceReader.setLinesToSkip(1);

		resourceReader.setLineMapper(new DefaultLineMapper<InputData>() {

			private DelimitedLineTokenizer lineTokenizer = new DelimitedLineTokenizer(";");
			private FieldSetMapper<InputData> fieldSetMapper = new BeanWrapperFieldSetMapper<InputData>() {
				@Override
			public InputData mapFieldSet(FieldSet pFielSet) throws BindException {
					InputData inputData = new InputData();
					inputData.setProductName(pFielSet.readString("productName"));
					inputData.setProductEanCode(pFielSet.readString("productEanCode"));
					inputData.setProductType(pFielSet.readString("productType"));
					inputData.setProductAmount(pFielSet.readDouble("productAmount"));
					inputData.setProductQuantity(pFielSet.readInt("productQuantity"));
					inputData.setSupplierName(pFielSet.readString("supplierName"));
					inputData.setSupplierAddress(pFielSet.readString("supplierAddress"));
					inputData.setPurchaserEmail(pFielSet.readString("purchaserEmail"));
					inputData.setPurchaserFirstName(pFielSet.readString("purchaserFirstName"));
					inputData.setPurchaserLastName(pFielSet.readString("purchaserLastName"));
					inputData.setTransactionDate(pFielSet.readDate("transactionDate"));
					return inputData;

			}
			};

			@Override
			public InputData mapLine(String pLine, int pLineNumber) throws Exception {

				lineTokenizer.setNames(new String[]{"productName", "productEanCode", "productType", "productQuantity",
								"productAmount", "supplierName", "supplierAddress", "purchaserFirstName",
								"purchaserLastName", "purchaserEmail", "transactionDate"});
				return fieldSetMapper.mapFieldSet(lineTokenizer.tokenize(pLine));
			}

		});

		return resourceReader;
	}

}
```

Avec InputData égale à :

Sélectionnez

```
1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
24.
25.
26.
27.
28.
29.
30.
31.
32.
33.
34.
35.
36.
37.
38.
39.
```

```
package com.gkemayo.batch.dto;

import java.util.Date;

import lombok.EqualsAndHashCode;
import lombok.Getter;
import lombok.RequiredArgsConstructor;
import lombok.Setter;
import lombok.ToString;

@Setter
@Getter
@ToString
@EqualsAndHashCode
@RequiredArgsConstructor
public class InputData {

	private String productName;

	private String productType;

	private String productEanCode;

	private Double productAmount;

	private Integer productQuantity;

	private String purchaserFirstName;

	private String purchaserLastName;

	private String purchaserEmail;
```

```
        private String supplierName;

        private String supplierAddress;

        private Date transactionDate;
}
```

**Explications**

Spring Batch offre plusieurs classes de type Reader fournissant des fonctionnalités permettant de répondre à certains besoins de base : le besoin lire des données dans une base de données, le besoin des données dans un fichier plat, le besoin de lire des données dans plusieurs fichiers plats à la fois, le besoin de lire des données dans un fichier XML, etc. Voici quelques-unes de ces
classes : **ItemReader**, **FlatFileItemReader**, **MultiResourceItemReader**, **StaxEventItemReader**, **JdbcPagingItemReader**, **JpaPagingItemReader**, **JmsItemReader**, **JsonIte**
etc. Pour plus de détails, regarder le lien suivant.

Dans notre exemple et conformément au besoin métier décrit à la section III, nous avons besoin de lire dans plusieurs fichiers plats (de type csv) à la fois. C'est pourquoi notre classe **BatchReader** étend naturellement la classe **MultiResourceItemReader**. Les développements effectués dans notre classe **BatchReader** sont des développements qui suivent le canevas conventionnel tracé par Spring Batch, à savoir définir ou créer des objets permettant de parser chaque fichier d'entrée et de transformer chaque ligne lue en un objet Java (ici, **InputData**). On indique ainsi, le délimiteur/séparateur (ici le **;**) entre les différents champs d'une ligne. L'ordre de valorisation des propriétés de InputData est important, il va de la gauche vers la droite pour chaque ligne lue.

## V-F. La classe Processor▲

Sélectionnez
1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
24.
25.
26.
27.
28.
29.
30.
31.
32.
33.
34.
35.
36.
37.
38.
39.

40.
41.

```
42.
43.
44.
45.
46.
47.
48.
49.
50.
51.
52.
53.
54.
55.
56.
57.
58.
59.
60.
61.
62.
63.
64.
65.
66.
67.
68.
69.
70.
71.
72.
73.
74.
75.
76.
77.
78.
79.
80.
```

```
package com.gkemayo.batch.processor;

import org.springframework.batch.item.ItemProcessor;
import org.springframework.beans.factory.annotation.Autowired;

import com.gkemayo.batch.dto.ConvertedInputData;
import com.gkemayo.batch.dto.InputData;
import com.gkemayo.batch.entities.Order;
import com.gkemayo.batch.entities.Product;
import com.gkemayo.batch.entities.PurchaseDate;
import com.gkemayo.batch.entities.Purchaser;
import com.gkemayo.batch.entities.Supplier;
import com.gkemayo.batch.entitiesrepository.ProductRepository;
import com.gkemayo.batch.entitiesrepository.PurchaseDateRepository;
import com.gkemayo.batch.entitiesrepository.PurchaserRepository;
import com.gkemayo.batch.entitiesrepository.SupplierRepository;

public class BatchProcessor implements ItemProcessor<InputData, ConvertedInputData> {

	@Autowired
	private SupplierRepository supplierRepository;

	@Autowired
	private PurchaserRepository purchaserRepository;

	@Autowired
	private ProductRepository productRepository;

	@Autowired
	private PurchaseDateRepository purchaseDateRepository;

	@Override
```

```
	public ConvertedInputData process(InputData item) throws Exception {

		ConvertedInputData convertedInputData = new ConvertedInputData();
		Order order = new Order();

		Supplier supplier = supplierRepository.findByName(item.getSupplierName());
		Purchaser purchaser = purchaserRepository.findByEmail(item.getPurchaserEmail());
		Product product = productRepository.findByEanCode(item.getProductEanCode());
		PurchaseDate purchaseDate = purchaseDateRepository.findByDate(item.getTransactionDate());

		if(supplier == null) {
			supplier = Supplier.of(null, item.getSupplierName(), item.getSupplierAddress());
			 convertedInputData.setSupplier(supplier);
		}else {
			order.setSupplierId(supplier.getId());
		}

		if(purchaser == null) {
			purchaser = Purchaser.of(null, item.getPurchaserFirstName(), item.getPurchaserLastName(), item.getPurchaserEmail());
			convertedInputData.setPurchaser(purchaser);
		}else {
			order.setPurchaserId(purchaser.getId());
		}

		if(product == null) {
			product = Product.of(null, item.getProductName(), item.getProductType(), item.getProductEanCode());
			convertedInputData.setProduct(product);
			order.setQuantity(item.getProductQuantity());
			order.setAmount(item.getProductAmount()*item.getProductQuantity());
		}else {
			order.setProductId(product.getId());
			order.setQuantity(item.getProductQuantity());
			order.setAmount(item.getProductAmount()*item.getProductQuantity());
		}

		if(purchaseDate == null) {
			purchaseDate = PurchaseDate.of(null, item.getTransactionDate());
			convertedInputData.setPurchaseDate(purchaseDate);
		}else {
			order.setDateId(purchaseDate.getId());
		}

		convertedInputData.setOrder(order);

		return convertedInputData;
	}

}
```

Avec ConvertedInputData égale à :

Sélectionnez

```
20.
21.
22.
23.
package com.gkemayo.batch.dto;

import com.gkemayo.batch.entities.Supplier;
import com.gkemayo.batch.entities.Order;
import com.gkemayo.batch.entities.Product;
import com.gkemayo.batch.entities.Purchaser;
import com.gkemayo.batch.entities.PurchaseDate;

import lombok.Data;

@Data
public class ConvertedInputData {

        private Supplier supplier;

        private Purchaser purchaser;

        private Product product;

        private PurchaseDate purchaseDate;

        private Order order;
}
```

**Explications**

Spring Batch offre également plusieurs classes de type Processor en fonction des besoins : **ItemProcessor**, **AsyncItemProcessor**, **BeanValidatingItemProcessor**, **FunctionItemProcessor**, **ItemProcessorAdapter**, **PassThroughItemProcessor**, **ScriptIten** etc.
Dans notre exemple, nous effectuons un traitement simple sur les données lues. Notre classe **BatchProcessor** implémente donc simplement l'interface de base Spring Batch **ItemProcessor**. Notre Processor se chargera d'agréger les données InputData lues par le Reader dans un autre objet ConvertedInputData dans lequel elles sont regroupées par entités (Product, Purchaser, etc.) en veillant à ne pas en créer une si elle existe déjà en base.

## V-G. La classe Writer▲

Sélectionnez

```
1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
24.
25.
26.
27.
28.
```

```
29.
30.
31.
32.
33.
34.
35.
36.
37.
38.
39.
40.
41.
42.
43.
44.
45.
46.
47.
48.
49.
50.
51.
52.
53.
54.
55.
56.
57.
58.
59.
60.
61.
62.
63.
```

```
package com.gkemayo.batch.writer;

import java.util.List;

import org.springframework.batch.item.ItemWriter;
import org.springframework.beans.factory.annotation.Autowired;

import com.gkemayo.batch.dto.ConvertedInputData;
import com.gkemayo.batch.entities.Supplier;
import com.gkemayo.batch.entities.Product;
import com.gkemayo.batch.entities.Purchaser;
import com.gkemayo.batch.entities.PurchaseDate;
import com.gkemayo.batch.entitiesrepository.SupplierRepository;
import com.gkemayo.batch.entitiesrepository.OrderRepository;
import com.gkemayo.batch.entitiesrepository.ProductRepository;
import com.gkemayo.batch.entitiesrepository.PurchaserRepository;
import com.gkemayo.batch.entitiesrepository.PurchaseDateRepository;

public class BatchWriter implements ItemWriter<ConvertedInputData> {

	@Autowired
	private SupplierRepository supplierRepository;

	@Autowired
	private PurchaserRepository purchaserRepository;

	@Autowired
	private ProductRepository productRepository;

	@Autowired
	private PurchaseDateRepository purchaseDateRepository;

	@Autowired
	private OrderRepository commandRepository;

	@Override
```

```
	public void write(List<? extends ConvertedInputData> items) throws Exception {
		items.stream().forEach(item -> {
			Supplier supplier = null;
			Purchaser purchaser = null;
			Product product = null;
			PurchaseDate purchaseDate = null;
			if(item.getOrder().getSupplierId() == null) {
				supplier = supplierRepository.save(item.getSupplier());
				item.getOrder().setSupplierId(supplier.getId());
			}
			if(item.getOrder().getPurchaserId() == null) {
				purchaser = purchaserRepository.save(item.getPurchaser());
				item.getOrder().setPurchaserId(purchaser.getId());
			}
			if(item.getOrder().getProductId() == null) {
				product = productRepository.save(item.getProduct());
				item.getOrder().setProductId(product.getId());
			}
			if(item.getOrder().getDateId() == null) {
				purchaseDate = purchaseDateRepository.save(item.getPurchaseDate());
				item.getOrder().setDateId(purchaseDate.getId());
			}
			commandRepository.save(item.getOrder());
		});
	}

}
```

**Explications**

Spring Batch offre enfin plusieurs classes de type Writer en fonction des besoins : **ItemWriter**, **HibernateItemWriter**, **FlatFileItemWriter**, **JmsItemWriter**, **JpaItemWriter**, **KafkaItemWriter**, **MongoItemWriter**, etc. Pour plus de détails, regarder le lien suivant.
Notre classe **BatchWriter** implémente l'interface de base **ItemWriter** et effectue des enregistrements platoniques en base.

## V-H. La classe SkipPolicy▲

Sélectionnez
1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
24.
25.
26.
27.
28.
29.

```
package com.gkemayo.batch.batchutils;
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.core.step.skip.SkipLimitExceededException;
import org.springframework.batch.core.step.skip.SkipPolicy;
import org.springframework.batch.item.file.FlatFileParseException;

import lombok.extern.slf4j.Slf4j;

@Slf4j
public class BatchStepSkipper implements SkipPolicy {

	Logger log = LoggerFactory.getLogger(BatchStepSkipper.class);

	private static Integer SKIP_THRESHOLD = 3;

	@Override
	public boolean shouldSkip(Throwable t, int skipCount) throws SkipLimitExceededException {
		if(t instanceof FlatFileParseException && skipCount < SKIP_THRESHOLD) {
			log.info(((FlatFileParseException) t).getLineNumber() + ": " +((FlatFileParseException) t).getInput() + " -- " + ((FlatFileParseException) t).getMessage());
			return true;
		}
		return false;
	}



}
```

**Explications**

Pour qu'une classe soit vue par Spring Batch comme une classe de gestion de la tolérance aux fautes, il faut qu'elle implémente l'interface **SkipPolicy**. C'est le cas de notre classe **BatchStepSkipper**. Cette classe implémentera ainsi la méthode shouldSkip() dans laquelle on indiquera de tolérer au plus trois fois des erreurs de type FlatFileParseException dans un fichier d'entrées lu par le BatchReader.

## V-I. La classe Listener▲

Sélectionnez

```
1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
24.
25.
26.
```

```
package com.gkemayo.batch.batchutils;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.JobExecutionListener;
```

```
import lombok.extern.slf4j.Slf4j;

@Slf4j
public class BatchJobListener implements JobExecutionListener {

	Logger log = LoggerFactory.getLogger(BatchJobListener.class);

	@Override
	public void beforeJob(JobExecution jobExecution) {
		log.info("Begining of job  " + jobExecution.getJobInstance().getJobName() + " at " + jobExecution.getStartTime());
	}

	@Override
	public void afterJob(JobExecution jobExecution) {
		String exitCode = jobExecution.getExitStatus().getExitCode();
		log.info("End of job  " + jobExecution.getJobInstance().getJobName() + " at " + jobExecution.getEndTime() + " with status " + exitCode);
	}

}
```

**Explications**

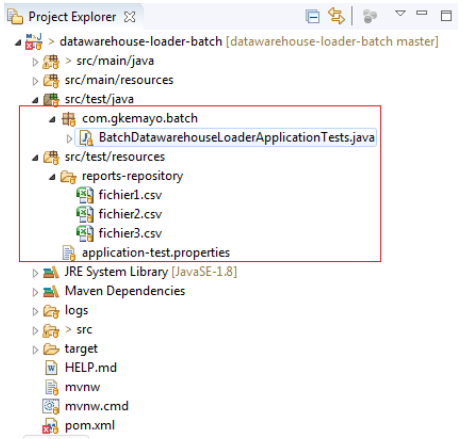
Spring Batch propose plusieurs types de Listener pour chacun de ces composants. On peut citer le **JobExecutionListener**, **StepExecutionListener**, **ItemReadListener**, **ItemProcessListener**, **ItemWriteListener**, **SkipListener**. Pour rappel, un Listener Spring Batch est une Interface qui offre des méthodes standards qui permettront au développeur d'ajouter sa propre implémentation à l'effet d'observer et de journaliser les évènements qui surviennent dans un composant durant l'exécution d'une instance du Batch. Son utilisation est facultative.
Pour notre exemple, nous avons implémenté la classe **BatchJobListener** qui sera associée au **Job**du Batch. Elle a pour but de dire à quelle heure le job s'est démarré, et dans quel état et à quelle heure il s'est terminé.

## VI. Le Test Unitaire/Intégration▲

Le test unitaire est une procédure qui permet de vérifier que le fonctionnement de chaque méthode développée dans un programme correspond aux besoins attendus. Quant au test d'intégration, il permet de vérifier le bon fonctionnement de l'intercommunication entre différents composants intégrés ensemble dans le programme. Pour y arriver, en général il faut mettre en place des configurations donnant la possibilité de réaliser ces tests. Spring Boot comme Spring Batch offrent des outils qui facilitent l'écriture de tests. Nous allons réaliser un petit test d'intégration pour notre exemple. Nous vous laissons le soin de vous en inspirer pour aller plus loin.

## VI-A. Configurations requises▲

Pour faire fonctionner notre test, la capture ci-dessous montre à travers le rectangle au trait rouge les configurations que nous allons réaliser :




### VI-A-1. application-test.properties▲

Sélectionnez
1.
2.
3.

```
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
############# DataSource Config #################
spring.datasource.name=datawarehouse-db-test
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.url= jdbc:h2:mem:datawarehouse-db-test
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.sql-script-encoding= UTF-8

spring.datasource.schema = classpath:schema/datawarehouse-schema.sql

spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
#http://localhost:8080/h2-console/

path.to.the.work.dir=./src/test/resources/reports-repository
```

**Explicatioons** :

Par défaut, Spring Boot en environnement de test chargera le contexte Spring de l'application (à savoir la classe BatchDatawarehouseLoaderApplication), mais aussi le fichier de configuration application.properties que nous avons présenté précédemment. Une conséquence de laisser les choses telles qu'elles, sera que vous pourrez via votre test laisser écrire des données pourries dans votre base de données de production. C'est pourquoi Spring Boot donne la possibilité de définir un fichier de configuration propre au test qui conventionnellement est défini sous le format application-**yyy**.properties. **yyy** correspond à ce qu'on appelle un profile et dans notre cas, il est égal à **test**.

Dans notre fichier **application-test.properties** ci-dessus, nous redéfinissons presque tous les paramètres utiles qui ont été déclarés dans l'application.properties. On peut y retenir deux choses : l'utilisation d'une base de données mémoire H2 (datawarehouse-db-test) pour notre test, et la définition d'un dossier devant contenir les jeux de données de notre test, à savoir quelques fichiers que doit lire le batch. Ce dossier est placé directement dans le classpath de l'application pour lui permettre de le rejouer autant de fois que possible sans aucun problème de portabilité (./src/test/resources/reports-repository).

### VI-A-2. Le dossier reports-repository▲

Il contient comme on peut le voir sur la figure de la section VI-A, nos trois fichiers de JDD donc voici le contenu :

Fichier 1 (contenant deux lignes en erreur):

Sélectionnez

```
1.
2.
3.
4.
PRDT_NAME;PRDT_EAN_CODE;PRDT_TYPE;PRDT_QTE;PRDT_AMOUNT;SUPPLIER_NAME;SUPPLIER_ADDRESS;PURCHASER_FIRST_NAME;PURCHASER_LAST_NAME;PURCHASER_EMAIL;TRANSACTION_DATE
ZenPhone5;EAN2918098;Phone;2;300.0;ASUS;7 Place Vendome Paris France;Eric;Dupont;ericdupont@yopmail.com;2019-12-18T17:09:42.411
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
YYYYYYYYYYYYYYYYYYYY
```

Fichier 2 :

Sélectionnez

```
1.
2.
PRDT_NAME;PRDT_EAN_CODE;PRDT_TYPE;PRDT_QTE;PRDT_AMOUNT;SUPPLIER_NAME;SUPPLIER_ADDRESS;PURCHASER_FIRST_NAME;PURCHASER_LAST_NAME;PURCHASER_EMAIL;TRANSACTION_DATE
ZenPhone5;EAN2918098;Phone;2;300.0;ASUS;7 Place Vendome Paris France;Eric;Dupont;ericdupont@yopmail.com;2019-12-15T17:09:42.411
```

Fichier 3 :

Sélectionnez

```
1.
```

```
2.
3.
PRDT_NAME;PRDT_EAN_CODE;PRDT_TYPE;PRDT_QTE;PRDT_AMOUNT;SUPPLIER_NAME;SUPPLIER_ADDRESS;PURCHASER_FIRST_NAME;PURCHASER_LAST_NAME;PURCHASER_EMAIL;TRANSACTION_DATE
USBZEN;EAN10000;USB KEY;8;15,99;Cdiscount;27 Boulevard des Nations Nantes France;Eric;Dupont;ericdupont@yopmail.com;2019-12-15
CameraZEN;EAN5000;Camera;12;160,57;Cdiscount;27 Boulevard des Nations Nantes France;Samuel;Marimont;samuelmarimont@yopmail.com;2019-12-15
```

## VI-B. La classe de test▲

Sélectionnez

```
1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
24.
25.
26.
27.
28.
29.
30.
31.
32.
package com.gkemayo.batch;

import org.junit.Assert;
import org.junit.jupiter.api.Test;
import org.junit.runner.RunWith;
import org.springframework.batch.core.BatchStatus;
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.boot.test.context.SpringBootTest;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.TestPropertySource;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
@SpringBatchTest
@ActiveProfiles("test")
@TestPropertySource(locations = "classpath:application-test.properties")
class BatchDatawarehouseLoaderApplicationTests {

	@Autowired
	JobLauncherTestUtils jobLaucherTestUtils;

	@Test
	public void testBatchDatawarehouseLoader() throws Exception {
		JobExecution jobExecution = jobLaucherTestUtils.launchJob();
```

```
                Assert.assertEquals(BatchStatus.COMPLETED, jobExecution.getStatus());
        }
}
```

**Explications**

- **@RunWith(SpringRunner.class) :** invoque le runner SpringRunner pour notre classe de test.
- **@SpringBootTest :** utilisé en environnement Spring Boot et invoque dans notre classe de test : le contexte d'application Spring, JUnit, les outils de bouchonnage tels que MockBean, etc.
- **@SpringBatchTest :** donne la possibilité d'utiliser dans notre classe de test, des objets Spring Batch conçus pour faire du test tel JobLauncherTestUtils. JobLauncherTestUtils est une classe Spring Batch disposant des fonctionnalités permettant de lancer dans les conditions réelles un Batch dans un test d'intégration.
- **@ActiveProfiles(« test ») :** active le profile test pour notre la classe de test.
- **@TestPropertySource(locations = « classpath:application-test.properties ») :**permet que le contexte d'application Spring de notre test utilise les propriétés contenues dans le fichier application-test.properties.

## VII. Démo de l'application▲

## VIII. Accéder au code source▲

Le code source de l'application se trouve ici : https://gitlab.com/gkemayo/datawarehouse-loader-batch.

## IX. Conclusion et perspectives▲

Il existe dans ce forum, d'excellents tutoriels qui ont déjà écrit sur le framework Spring Batch. Vous pouvez les consulter ici. Vous pourrez d'ailleurs compléter les notions que vous aurez apprises dans cet article en lisant celles de ces tutoriels. Chacun d'entre eux présente certains concepts précis du framework que nous n'avons pu aborder ici. Notamment par exemple, l'autonettoyage de Spring Batch, la notion de Tasklet, la parallélisation des traitements Spring Batch, le scheduling automatisé de Batch, etc. Dans ces différents tutoriels, vous apprendrez comment utiliser Spring Batch en dehors de Spring Boot, ce qui est un peu l'ancienne école dans un monde où Spring Boot a tendance à devenir incontournable dans l'écosystème Spring.

Au terme de notre article donc, nous vous avons présenté comment utiliser Spring Batch via Spring Boot pour résoudre un problème de traitement par lot sur un exemple d'application. Notre souci majeur avant tout a été d'abord de vous montrer comment on schématise conceptuellement une problématique de traitement par lot et comment Spring Batch se positionne dans ce schéma (cf. figure de la section II). Cela parce que nous avons remarqué que beaucoup de tutoriels existants autour de Spring Batch, entrent directement dans les aspects techniques du framework sans jamais se préoccuper de l'immersion du lecteur dans la problématique du traitement par lot. Nous pensons que c'est un élément important qui donne les armes mentales à quelqu'un qui débute de garder un fil conducteur. Nous vous avons aussi présenté comment utiliser Spring Batch dans l'écosystème Spring Boot afin de tirer profit des avantages de ce dernier qui peuvent se résumer à l'autoconfiguration et l'amélioration de la productivité. Enfin, nous rappelons que toutes les configurations de beans sont faites en JavaConfig et non en XML. Ce qui est plutôt la tendance actuellement.

Pour finir et comme nous le disions, nous avons utilisé un exemple d'application pour guider cet article. Un exemple d'application ne pouvant permettre d'aborder de façon exhaustive les contours de Spring Batch, nous nous sommes focalisés sur les fonctionnalités du framework qui permettent de résoudre notre problème. Nous avons mis en avant la structuration

d'un batch construit avec Spring Batch, nous avons donné les avantages et les inconvénients de ce dernier, nous avons présenté théoriquement et techniquement tous les concepts Spring Batch utilisés dans cet article en tâchant de faire un petit état de l'art d'autres concepts connexes non abordés dans cet article. Notamment sur les différentes classes d'implémentation possibles des interfaces ItemReader, ItemProcessor et ItemWriter. Ensuite, notre exemple nous a amenés à aborder de façon succincte la notion d'entrepôt de données (data warehouse) que vous pourrez approfondir au travers du lien de référence donné dans la section suivante. Nous avons également profité pour vous montrer l'utilisation des frameworks tels que Lombok ou encore Spring Data JDBC qui ont concouru à la construction complète de notre application. Nous avons terminé en présentant comment construire un test d'intégration de notre projet Spring Batch.

## X. References▲


- https://spring.io/projects/spring-batch
- https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-documentation
- https://gkemayo.developpez.com/tutoriels/java/tutoriel-sur-creation-application-web-avec-angular7-et-spring-boot-2/?page=creation-du-projet-spring-boot#LIII
- https://docs.spring.io/spring-batch/docs/current/reference/html/schema-appendix.html
- https://www.cartelis.com/blog/data-warehouse-modelisation-etoile/
- Spring Data JDBC : https://spring.io/blog/2018/09/17/introducing-spring-data-jdbc
- Spring Data JPA : https://docs.spring.io/spring-data/jpa/docs/1.5.0.RELEASE/reference/html/repositories.html
- Appendix A. Common application properties
- https://docs.spring.io/spring-batch/docs/current-SNAPSHOT/reference/html/appendix.html


## XI. Remerciements▲


Je voudrais dire un grand merci à :

- Mickael Baron pour sa relecture technique et pour ses remarques
- Claude Leloup et à escartefigue pour la relecture orthographiqu