Cahier de TP
« Spring Batch»

Pré-requis :

- Bonne connexion Internet
- Système d'exploitation recommandé : Linux, MacOs, Windows 10 (avec Git Bash)
- JDK8+
- IDE Recommandés : STS 4, IntelliJIDEA, VSCode avec accès aux dépôts de MavenCentral
- Outils optionnels recommandés : Docker, Git
- Accès à une Base de données (Postgres de préférence, soit par docker, soit par une installation spécifique)

docker-compose -f postgres-docker-compose.yml up -d

Atelier 1: Starter SpringBatch

1. Beans de @EnableBatchProcessing

Créer un Spring Starter Project, choisir Maven et ***org.formation*** comme *groupId* et *package racine*, déclarer ensuite les starters suivants :

- Spring Batch
- H2

Activer la configuration par défaut via l'annotation ***@EnableBatchProcessing***

Visualiser les beans créés au démarrage par cette annotation

2. Mise en place BD postgres

Démarrer un serveur Postgres.

Créer une base ***spring-batch***

Ajouter le starter Postgres

Configurer l'application afin qu'elle utilise la Base Postgres et qu'elle initialise le schéma dans un profil SpringBoot ***prod***

Visualiser les tables générées

Atelier 2: Premiers Jobs avec SpringBatch

2.1. Configuration basique

Récupérer les sources fournis (Un ItemReader, un ItewWriter, 1 classe de modèle et 1 classe de configuration)

Compléter la classe de configuration afin de configurer un job avec 2 steps qui utilisent les *ItemReaders/Writers* fournis et qui configure un listener

Le listener affiche juste sur la console le contexte d'exécution du job

Démarrer l'application SpringBoot et vérifier le démarrage automatique du job

Visualiser les tables Postgres

Essayer de redémarrer le job

Vous pouvez utiliser le script *drop_schema.sql* pour réinitialiser la base

2.2 Fichier à plat

Récupérer le fichier tabulé fourni, ils représentent une liste de produit associé à 3 fournisseurs différents.

Implémenter un job Batch qui lit le fichier en entrée et écrit 1 fichiers en sortie.

Le fichier en entrée correspond à la classe fournie ***org.formation.model.InputProduct***

La classe utilise des contraintes de Java Bean Validation, il faut ajouter le starter validation afin qu'elle compile.

Le fichier en sortie est une fichier qui ne reprend que les champs ***reference, nom, hauteur, largeur, longueur*** et ***fournisseurId***

Méthodologie :

- Modifier la configuration du job afin qu'il ne contienne qu'une étape : ***productStep***
- Définir la step ***productStep*** avec un ***productReader*** et un ***productWriter***, définir la taille du ***chunk***
- Implémenter ***productReader*** avec un ***FlatFileItemReader*** constitué de :
    - ***defaultLineMapper*** lui même constitué de
        - ***DelimitedLineTokenizer***
        - ***BeanWrapperFieldSetMapper***
- Implémenter ensuite productWriter avec un ***FlatFileItemWriter*** constitué de

- ***BeanWrapperFieldExtractor***
- et configuré en mode ***délimité*** (csv)

2.3 ItemProcessor

Insérer dans la step précédente une *CompositeItemProcessor* qui permettra :

- Filtrer les valeurs invalides avec Java Bean Validation
- Filtrer les produits pour ne conserver que les produits du fournisseur 1
- Transformer les classes *InputProduct* en la classe *OutputProduct* fournie

2.4 XML, JSON

Reprendre le fichier json fourni qui représente quasiment la même liste de produits

Changer la configuration du job précédent afin de lire à partir de ce fichier et écrire au format XML en utilisant JaxB2

Attention si vous êtes en Java 11, on doit explicitement ajouter la dépendance pour Jaxb2 :

```
<dependency>
	<groupId>javax.xml.bind</groupId>
	<artifactId>jaxb-api</artifactId>
	<version>2.3.0</version>
</dependency>
<dependency>
	<groupId>org.glassfish.jaxb</groupId>
	<artifactId>jaxb-runtime</artifactId>
	<version>2.3.2</version>
	<scope>runtime</scope>
</dependency>
<dependency>
	<groupId>com.sun.activation</groupId>
	<artifactId>jakarta.activation</artifactId>
	<version>1.2.1</version>
	<scope>runtime</scope>
</dependency>
```

2.5 Base de données

Dans Postgres, créer une base ***produit*** et utiliser le script SQL fourni pour l'alimenter

Récupérer la classe ***org.formation.BDConfiguration*** qui définit les datasources utilisées pour cet atelier.

Configurer l'application en accord avec votre environnement et faire un premier test de démarrage afin de vérifier que cette classe n'a pas cassé la configuration précédente.

Implémenter un Job qui lit tous les produits du fournisseur 1 dans la base à l'aide d'un ***`JdbcPagingItemReader`*** et génère un fichier à plat (vous pouvez réutiliser le *writer* des TPS précédents)

Atelier 3: Configuration de Job

3.1. Démarrage de job

Désactiver le démarrage automatique des Jobs de SpringBoot

Implémenter l'interface ***CommandLineRunner*** dans la classe Main

Dans la méthode *run()*, exécuter le job avec comme paramètre :

- La date d'exécution

Tester des lancements successifs

Construire le jar et démarrer le programme en ligne de commande

3.2 Explorer de données

Récupérer le nouveau projet Maven fourni.

Il s'agit d'une application Web qui a pour dépendances :

- starter-web
- Postgres
- Thymeleaf, Bootstrap
- spring-batch

Compléter la classe du contrôleur répondant à la page d'accueil. Elle doit renseigner l'attribut jobs avec une List de ***JobDto*** avant de déléguer la requête à la page thymeleaf générant la vue

Atelier 4: Configuration de Step

1. Configuration Redémarrage

Configurer le `BeanValidatingItemProcessor` afin qu'il lance des exceptions de validation plutôt que de filter les items

Rendre le paramètre de Date non-identifiant

Configurer un job à 3 steps :

- La première lit le fichier *csv* en entrée et génère un fichier XML.
  Autoriser un maximum de 200 erreurs de validations
- La deuxième est identique mais faire en sorte que cette étape se ré-exécute quelque soit son statut
- La troisième étape lit également le fichier *csv* et génère également le fichier XML en sortie
  Aucune exception n'est tolérée
  Cette étape pourra s'exécuter au maximum 2 fois

Faire 3 démarrages, vous devez observer :

- Au premier étage, les étapes 1 et 2 réussissent. La troisième échoue à cause d'erreurs de validation
- Au second démarrage, l'étape 2 se ré-exécute. La troisième échoue à cause d'erreurs de validation
- Au 3ème démarrage, l'étape 2 se ré-exécute. La troisième ne s'exécute pas car elle a atteint sa limite

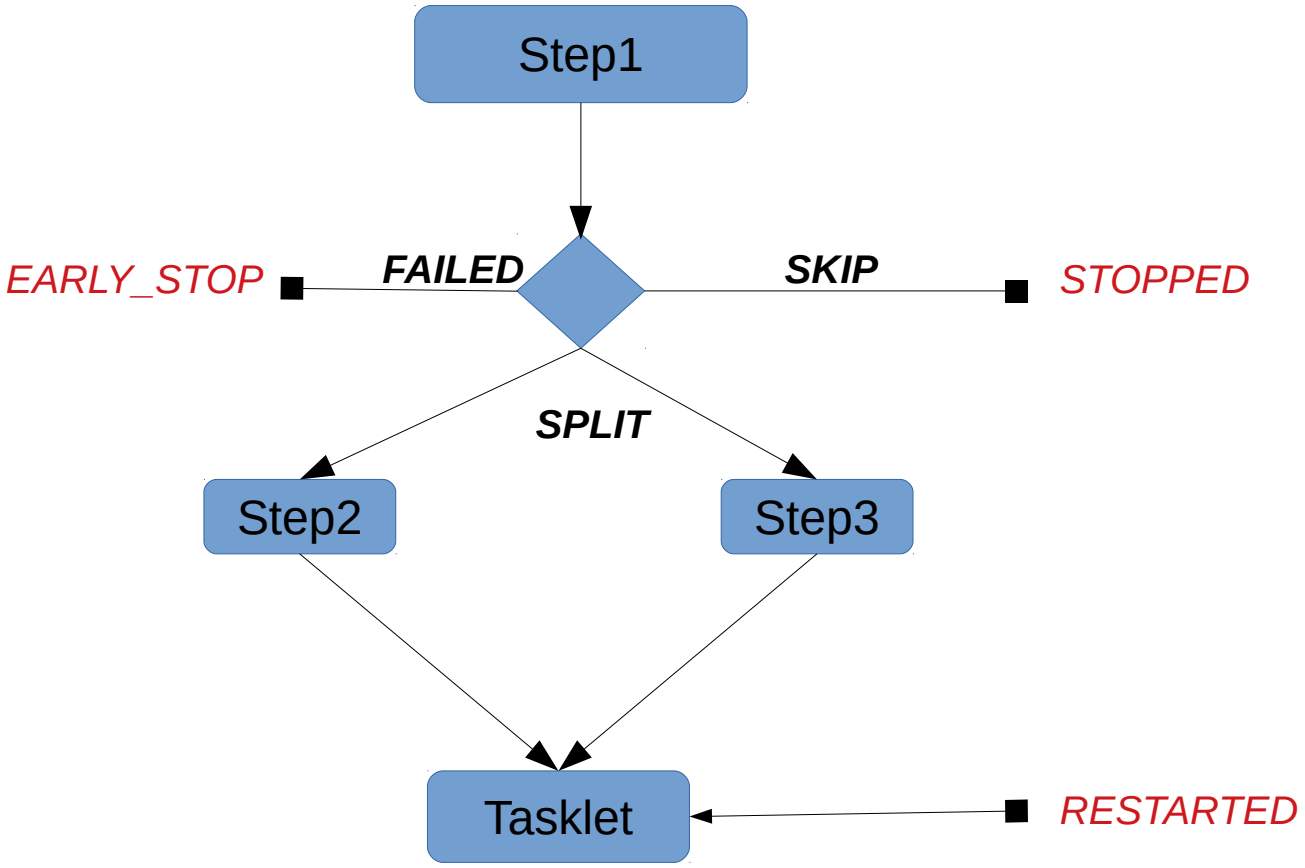
2. Listeners et Tasklet

Utiliser les annotations pour être prévenus des éléments ayant été ignorés en lecture

Implémenter une *TaskletStep* et l'insérer en dernier dans le flow séquentiel des étapes.

3. Flow

Configurer le flow suivant (En gras les *ExitStatus*, en rouge les *BatchStatus* du job)



Implémenter dans un premier temps un *ItemListener* afin que Step1 initialise son statut de sortie à ***SPLIT***

Faire un premier lancement pour valider le flow sans erreur

Dans un second temps, implémenter un *JobExecutionDecider* qui retourne un statut de sortie en fonction d'un paramètre de Job

Désactiver le listener précédent et modifier la configuration du Job

Exécuter l'application et vérifier le bon fonctionnement en passant en paramètre les différents codes de statut (*FAILED, SKIP , SPLIT*)

Atelier 5. Pour aller + loin

5.1 Partitionning

Le but est de partitionner le traitement d'importation de produits en fonction des fournisseurs afin d'alimenter une nouvelle table dans la base ***produit***

Créer une nouvelle table dans la base ***produit*** avec le script SQL fourni

Implémenter un partionneur qui définit 3 partitions dans chaque partition l'id du fournisseur est positionné dans le contexte

Configurer le job pour utiliser ce partitioneur

Pour l'écriture, utiliser le `JdbcBatchItemWriter` fourni

5.2 Tests unitaires

Reprendre le TP sur les fichiers JSON et XML, récupérer les 2 fichiers fourni (1 fichier d'entrée et un fichier de sortie)

Modifier le TP afin que le fichier d'entrée et celui de sortie soit spécifié par des paramètres d'entrée

Désactiver le démarrage automatique de job et modifier la classe principale afin qu'elle implémente l'interface ***CommandLineRunner*** pour démarrer le job en passant les paramètres

Écrire 2 méthodes de test :

- 1 démarrant le job complet
- 1 autre démarrant la step du Jobs

Ces 2 méthodes positionneront les paramètres des fichiers tests avant l'exécution du job ou de la step