Cahier de TP
« Spring Batch »

Outils utilisés :

- Bonne connexion Internet
- Système d'exploitation recommandé : Linux, MacOs, Windows 10 (avec Git Bash)
- JDK 1.5+
- 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

Dépôts github de référence :

- Support et Cahier de TPs : https://github.com/dthibau/springbatch.git
- Solutions Ateliers : https://github.com/dthibau/springbatch-solutions.git

Atelier 1: Beans SpringBatch

1.1 Contexte d'un job

Récupérer les sources du répertoire TP2.1/1_Beans

Visualiser le fichier de configuration ***simple-job-launcher-context.xml*** et les beans qui y sont définis

Exécuter la classe de test qui vérifie le chargement du context Spring méthode de test ***contextLoad()***

1.2 Mise en place BD MySQL

Démarrer un serveur MySQL.

Créer une base ***springbatch***

Exécuter la classe de test qui vérifie le chargement du context Spring et initialise la base MySQL ***setupMySQL()***

Visualiser les tables générées

Atelier 2: Premiers Jobs avec SpringBatch

2.1 Configuration basique

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

Mettre au point le fichier ***src/main/resources/jobs/dummyJob*** afin de configurer un job avec 2 steps qui utilisent les ItemReaders/Writers fournis et qui configure un listener de job

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

Créer une configuration de démarrage :

- Classe principale :
  org.springframework.batch.core.launch.support.CommandLineJobRunner
- Arguments :
  batch/dummy-job-context.xml dummyJob
- VM Arguments :
  -DENVIRONMENT=mysql

Commenter le bean d'initialisation de la base si votre base est déjà initialisée.

Lancer cette configuration de démarrage

Visualiser les tables MySQL

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***

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

Méthodologie :

- Créer un job ne contenant 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)

Se créer une configuration de démarrage pour tester ce job.

2.3 ItemProcessor

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

- Filtrer les valeurs invalides un Produit d'entrée doit respecter :
  - Champ ***reference*** non vide comprise entre 1 et 5 caractères
  - Champ ***nom*** non vide
- Filtrer les produits pour ne conserver que les produits du fournisseur 1
- Transformer les items *InputProduct* en la classe *OutputProduct* fournie

2.4 XML

Changer la configuration du job précédent afin d'écrire au format XML en utilisant XSTream

Le format de sortie attendu est :

```
<produit>
  <reference>REF1</reference>
  <hauteur>222.0</hauteur>
  <importedDate>2022-01-18T14:38:26.152092630Z</importedDate>
  <largeur>476.0</largeur>
  <longueur>457.0</longueur>
  <nom>3O3O6PR4YS98QZR</nom>
</produit>
```

2.5 Base de données

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

Récupérer le fichier ***domain-data-source-context.xml*** qui définit la datasource vers la base produits.

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 ou XML

Atelier 3 : Configuration de Job

3.1 Démarrage de job

Créer une classe principale Main qui dans la méthode main :

- Instancie le contexte Applicatif Spring
- Récupère un jobLauncher et le Job
- Démarre le job précédent en ajoutant un paramètre DATE dont la valeur sera new Date()

Tester des lancements successifs

Atelier 4. Configuration de Step

4.1 Configuration Redémarrage

Configurer le ***ValidatingItemProcessor*** afin qu'il lance des exceptions de validation plutôt que de filtrer 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

4.2 Scopes , Listeners et Tasklet

Configurer le job de la manière suivante :

- 1 ère étape qui prend en entrée un CSV et fourni en sortie un autre CSV
- 2nde étape qui prend en entrée la base de données et ajoute les produits au CSV précédent
- 3ème étape qui prend le CSV précédent et construit un XML de sortie

Les étapes 1 et 2 peuvent ignorer les données mal formées. L'étape 3 non.

Les éléments ignorés par les étapes 1 et 2 sont écrites dans les fichiers ***skip.csv*** et ***skip.json*** respectivement.

Le job définit 3 propriétés stockées dans le contexte d'exécution (elles pourront être lues à partir de *application.yml*)

- ***input.directory*** : Répertoire d'entrée ou sont présent les fichiers csv et json
- ***temp.directory*** : Répertoire temporaire ou est généré le fichier CSV temporaire
- ***output.directory*** : Répertoire de sortie ou est généré le fichier XML final

Les données (fichiers d'entrée/sortie de skip) sont variabilisées dans le contexte d'exécution du step et un listener positionne les données à partir des donnes du job

Implémenter ensuite 2 *TaskletStep* :

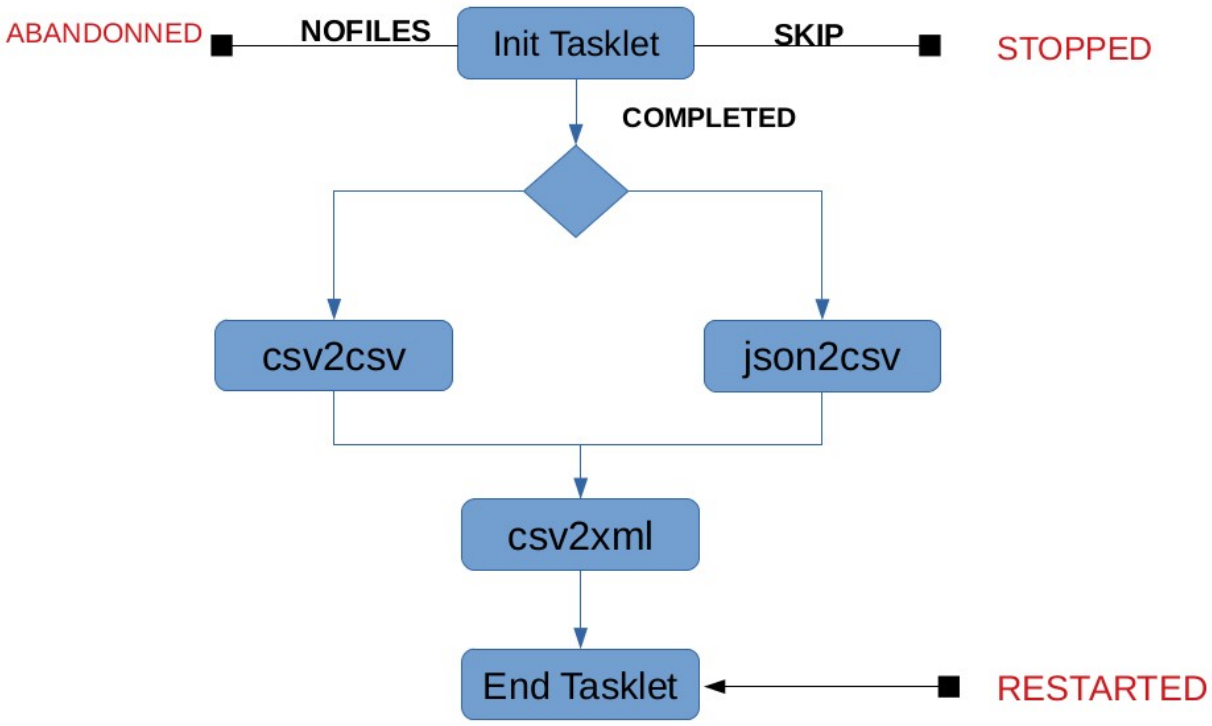
- La première en début de Job crée les répertoire ***inputDirectory***, ***tempDirectory*** et ***outputDirectory***, copie les fichiers d'entrée présents ailleurs dans *inputDirectory*.

Les répertoires de travail sont alors positionnés dans le contexte d'exécution du Job.

- La seconde en fin de job, nettoie les répertoires d'entrée et de travail.

4.3 Flow

On veut configurer le Step Flow suivant (En gras les ExitStatus, en rouge les BatchStatus du job)



La première étape responsable de recopier les fichiers sources dans le répertoire de travail *inputDirectory* peut renvoyer différents statuts :

- FAILED : Aucun fichier sources trouvés. (Exception lancée par l'étape InitStep)
- SKIP : Les fichiers sont trop anciens (on pourra utiliser un paramètre date pour les tests)

Faire un premier lancement pour valider le flow sans erreur

Tester ensuite le cas de l'exception

Finalement, implémenter un ***JobExecutionDecider*** qui positionne le statut SKIP artificiellement.

Tester alors le redémarrage

Optionnellement :

Pour traiter les problèmes de concurrence des étapes csv2csv et json2csv, écrire dans 2 fichiers séparés et utiliser un MultiResourceItemReader pour l’étape csv2xml

Atelier 5. Pour aller + loin

5.1 Partitionnement

Le but est de partitionner le traitement d'importation de produits en fonction des fournisseurs afin d'alimenter une nouvelle base de données

Créer une nouvelle base de données ***output_produits*** et une table ***new_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 un ***JdbcBatchItemWriter*** comme suit :

```
@Bean
@StepScope
public JdbcBatchItemWriter<InputProduct> jdbcProductWriter() {
  JdbcBatchItemWriter<InputProduct> itemWriter = new JdbcBatchItemWriter<>();
  itemWriter.setDataSource(outputProductDataSource);
  itemWriter.setSql("INSERT INTO NEW_PRODUIT (nom,reference) VALUES
(:nom, :reference)");
  itemWriter.setItemSqlParameterSourceProvider
  (new BeanPropertyItemSqlParameterSourceProvider<>());
  itemWriter.afterPropertiesSet();
}
return itemWriter;
```

5.2 Tests unitaires

Reprendre le TP sur les fichiers JSON et XML, récupérer les 2 fichiers fournis (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