Cahier de TP
« Spring Batch »

Outils utilisés :

- Bonne connexion Internet
- Système d'exploitation recommandé : Linux, MacOs, Windows 10 (avec Git Bash)
- JDK21+
- IDE Recommandés : STS 4, IntelliJIDEA, VSCode avec accès aux dépôts de MavenCentral
- Accès à une Base de données (Postgres de préférence, soit par docker, soit par une installation spécifique)
- Docker, Git

Dépôts github de référence :

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

Table des matières

Atelier 1: Starter SpringBatch

1.1 Beans de @EnableBatchProcessing

Créer un Spring Starter Project ***batch***, choisir Maven et ***org.formation*** comme *groupId* et package racine, ***batch*** comme *artifactId* et 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

1.2 Mise en place BD postgres

Démarrer un serveur Postgres. Si vous avez docker installé vous pouvez démarrer une base Postgres et un client pgAdmin

```
cd $TP_DATA
docker compose -f postgres-docker-compose.yml up -d
```

Créer une base ***spring-batch***

Ajouter le starter Postgres dans le projet ***batch***

Ajouter un profil de configuration SpringBoot ***prod***

Y définir la base Postgres et la propriété initialisant le schéma

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. Les 2 steps :

- utilisent les ItemReaders/Writers fournis
- travaillent avec un chunk de 10

Ajouter un listener qui affiche 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 fichier 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 : ***fileStep***
- Définir la step ***fileStep*** avec un ***productReader*** et un ***productWriter***, définir la taille du chunk
- Dans le package ***org.formation.io***, créer une classe de Configuration ***ReadersConfiguration***
- Y définir un bean ***productReader*** avec un ***FlatFileItemReader*** constitué de :
  - ***defaultLineMapper*** lui même constitué de
    - ***DelimitedLineTokenizer***
    - ***BeanWrapperFieldSetMapper***
- Dans le package ***org.formation.io***, créer une classe de Configuration ***WritersConfiguration***
- Y définir un bean ***productWriter*** avec un ***FlatFileItemWriter*** constitué de
  - Configuré en mode **délimité** (csv)
  - Ne reprenant que les champs précisés plus haut

2.3 ItemProcessor

Insérer dans la step précédente une classe ***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 items *InputProduct* en la classe *OutputProduct* fournie

Configurer cet *ItemProcessor* dans une classe ***org.formation.io.ProcessorsConfiguration***

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>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-json</artifactId>
</dependency>
<dependency>
     <groupId>org.springframework</groupId>
     <artifactId>spring-oxm</artifactId>
</dependency>
<dependency>
     <groupId>jakarta.xml.bind</groupId>
     <artifactId>jakarta.xml.bind-api</artifactId>
</dependency>
<dependency>
     <groupId>com.sun.xml.bind</groupId>
     <artifactId>jaxb-impl</artifactId>
     <scope>runtime</scope>
</dependency>
```

Le format de sortie attendu est :

```
<produit ref="7O717">
  <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 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 ce 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 :

- Un identifiant String

Tester des lancements successifs

Construire le jar et démarrer le programme en ligne de commande

3.2 Explorer les 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

4.1 Configuration Redémarrage

Configurer le ***BeanValidatingItemProcessor*** 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 un json 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 ignorées 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* :

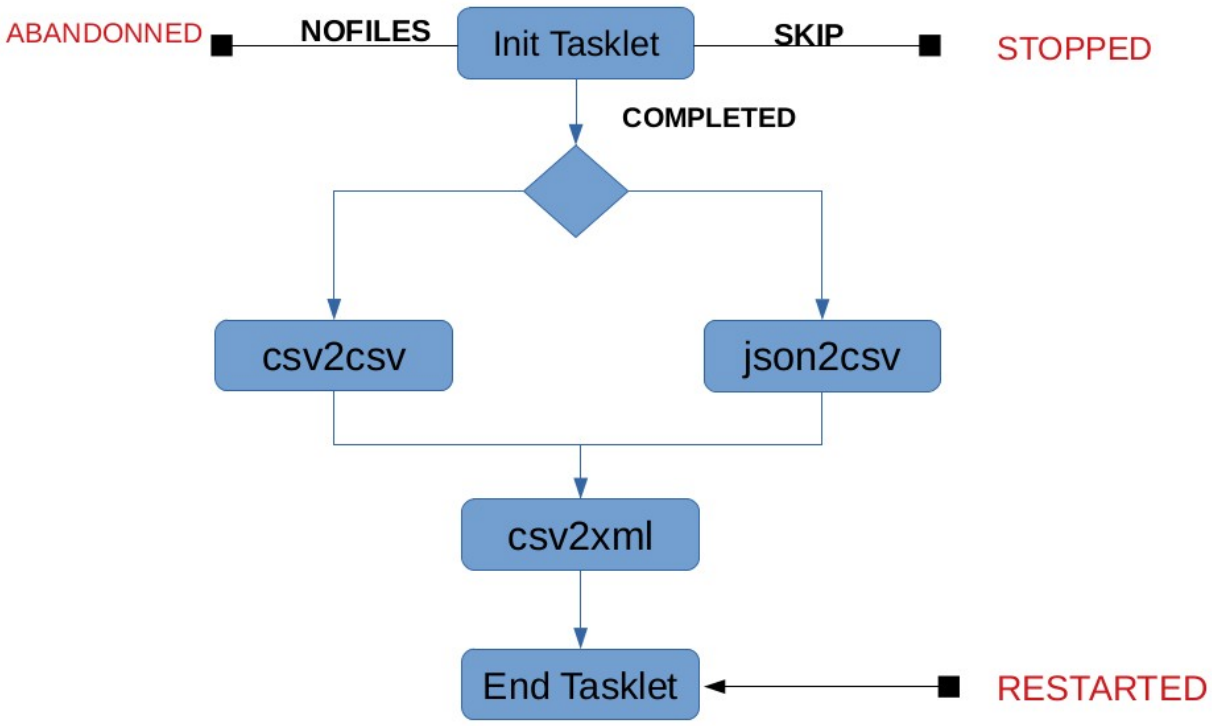
- La première en début de Job crée les répertoire ***inputDirectory***, ***tempDirectory*** et ***outputDirectory***, copie les fichiers d'entrée présents ailleurs dans *inputDirectory*.

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 ou le répertoire source est vide

Finalement, restaurer le répertoire des sources et implémenter un ***JobExecutionDecider*** qui positionne le statut SKIP artificiellement.

Le premier démarrage doit donner lieu à un statut STOPPED

Au redémarrage, le job doit reprendre à la tasklet de find (EndTaskLet)

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 Partitionning

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

Écrire 3 méthodes de test :

- 1 démarrant le job complet via ***launchJob*** et vérifiant le statut du job
- 1 démarrant la step initStep via ***launchStep*** et vérifiant son statut
- 1 dernier testant le MultiResourceReader, en créant manuellement un *StepExecutionContext* puis en utilisant ***StepScopeTestUtils.doInStepScope***