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
- 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

Visualiser les tables générées

Atelier 2: Configuration de Job

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. 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 2 paramètres :

- Le nombre d'itération. Récupérér la valeur dans les propriétés du projet SpringBoot (*application.properties/yml*)
- La date d'exécution (paramètre non identifiant)

Attention : Pour pouvoir utiliser le paramètre du nombre d'itération, décommenter le code fourni dans *DummyReader*

Construire le jar et démarrer le programme en ligne de commande

3. Démarrage de job

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 3: Configuration de Step

1. Configuration Redémarrage

Reprendre les sources fournis, il y a désormais 3 *ItemReader* sur le modèle de *DummyReader* :

- Le premier lance une Exception à la 100ème itération
- Le second ne lance jamais l'exception
- Le troisième lance une Exception à la 20ème itération

La configuration des paramètres a également changé, visualiser les changements

Configurer les 3 étapes dans le job de telle façon que :

- Dummy2 se réexécute quelquesoit son statut
- Dummy3 se réexécute au maximum 2 fois

Démarrer le job une première fois avec 100 itérations, puis le redémarrer avec 80 itérations et observer le comportement

Configurer ensuite l'étape3 afin quelle ignore un certain nombre d'exceptions

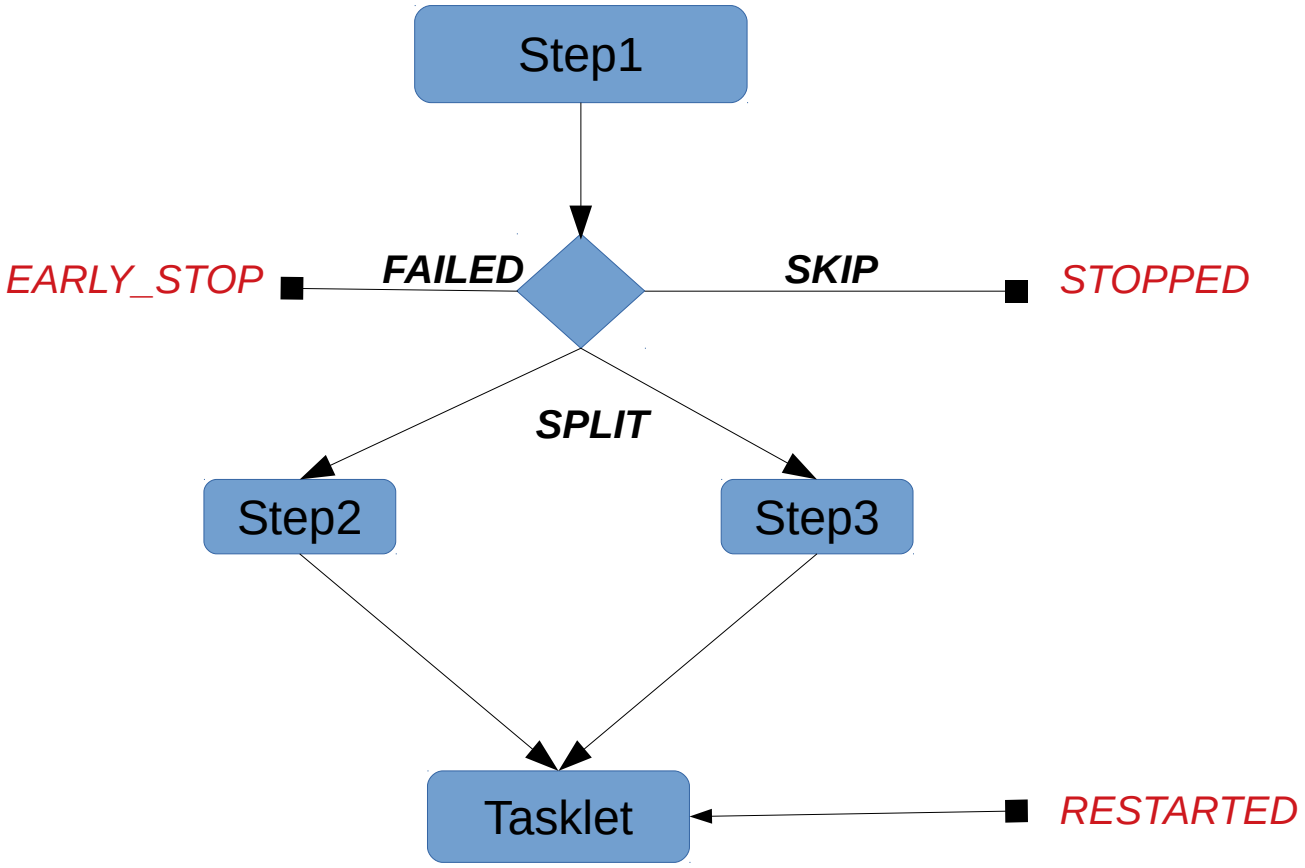
2. Listeners et Tasklet

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 4. ItemReader/ItemWriter

4.1 FlatFile

Créer un nouveau projet SpringBatch/SpringBoot avec les bonnes dépendances

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 sortie est une fichier dont chaque ligne correspond à un produit selon la classe ***org.formation.model.OutputProduct*** fourni, seul les produits du fournisseur 1 y sont écrits

N'oubliez pas de valider les entrées avec un ***BeanValidatingItemProcessor*** par exemple

4.2 XML, JSON

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

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>
```

4.2 Base de données

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

Créer un nouveau projet SpringBatch ***TP5***

Récupérer les classes ***org.formation.model.product*** et ***org.formation.BDConfiguration*** fournies

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 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)

Ecrire un cas de test qui teste en isolation l'unique step du job

Dans ce cas de test, fournir le fichier d'entrée et s'ssurer qu'il produit le fichier de sortie attendu

Atelier 5 : KafkaStream

Objectifs :
Écrire une mini-application Stream qui prend en entrée le topic ***position*** et écrit en sortie dans le topic ***position-out*** en ajoutant un timestamp aux valeurs d'entrée

Importer le projet Maven fourni, il contient les bonnes dépendances et un package *model* :

- Un champ timesatmp a été ajouté à la classe Courier
- Une implémentation de *Serde* permettant la sérialisation et la désérialisation de la classe *Courier* est fournie

Avec l'exemple du cours, écrire la classe principale qui effectue le traitement voulu

Atelier 6 : Fiabilité

1. At Least Once, At Most Once

Objectifs :

Explorer les différentes combinaisons de configuration des producteurs et consommateurs vis à vis de la fiabilité sous différents scénarios de test.

On utilisera un cluster de 3 nœuds avec un topic de 3 partitions et un mode de réplication de 2.

Le scénarios de test envisagé (Choisir un scénario parmi les 2):

- Redémarrage de broker(s)
- Ré-équilibrage des consommateurs

Les différentes combinaisons envisagées

- Producteur : *ack=0* ou *ack=all*
- Consommateur : auto-commit ou commits manuels

Les métriques surveillés

- Producteur : Trace WARN ou +
- Consommateur : Trace Doublon ou messages loupés

Méthodes :

Supprimer le topic ***position***

Le recréer avec

```
./kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 2 --partitions 3 --topic verify
```

Vérifier l'affectation des partitions et des répliques via :

```
./kafka-topics.sh --zookeeper localhost:2181 --describe --topic position
```

Récupérer les sources fournis et construire pour les 2 clients l'exécutable via

```
mvn clean package
```

Dans 2 terminal, démarrer 2 consommateurs :

```
java -jar target/consumer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 1 1000 >>
log1.csv
```

```
java -jar target/consumer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 1 1000 >>
log2.csv
```

Dans un autre terminal, démarrer 1 producteur multi-threadé :

```
java -jar target/producer-0.0.1-SNAPSHOT-jar-with-dependencies.jar 20 5000 10
<0|1|2> <0|all>
```

Pendant la consommation des messages, en fonction du scénario : arrêter et redémarrer un broker ou un consommateur.

Un utilitaire ***check-logs*** est fourni permettant de détecter les doublons ou les offsets.

```
java -jar check-logs.jar <log.csv>
```

2. Exactly One

Modifier la configuration du producer afin d'autoriser l'idempotence.

Rejouer les scénarios de tests précédents

Atelier 7 : Administration

Exécuter les producteurs et les consommateurs pendant les opérations d'administration

7.1 Reassign partitions, Retention

Modifier le nombre de partitions de position à 8

Vérifier avec

```
./kafka-topics.sh --zookeeper localhost:2181 --describe --topic position
```

Ajouter un nouveau broker dans le cluster.

Réexécuter la commande

```
./kafka-topics.sh --zookeeper localhost:2181 --describe --topic position
```

Effectuer une réaffectation des partitions en 3 étapes

Visualiser les segments et apprécier la taille

Diminuer le ***retention.bytes*** à une taille inférieure et visualiser les conséquences

7.2 Rolling restart

Sous charge, effectuer un redémarrage d'un broker.

Vérifier l'état de l'ISR

7.3 Mise en place de SSL

Travailler dans un nouveau répertoire ***ssl***

```
#  Créer son propre CA (Certificate Authority)
openssl req -new -newkey rsa:4096 -days 365 -x509 -subj "/CN=localhost" -keyout ca-key -out ca-cert -nodes
# Générer une paire clé publique/privé pour chaque serveur
keytool -keystore server.keystore.jks -alias localhost -validity 365 -genkey -keyalg RSA -storetype pkcs12
# Create Certificate signed request (CSR):
keytool -keystore server.keystore.jks -certreq -file cert-file -storepass secret -keypass secret -alias localhost
# Générer le CSR signé avec le CA
openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-file -out cert-file-signed -days 365 -CAcreateserial -passin pass:secret
# Importer le certifcat CA dans le KeyStore serveur
keytool -keystore server.keystore.jks -alias CARoot -import -file ca-cert -storepass secret -keypass secret -noprompt
# Importer Signed CSR dans le KeyStore
keytool -keystore server.keystore.jks -import -file cert-file-signed -storepass secret -keypass secret -noprompt -alias localhost
# Importer le certificat CA dans le TrustStore serveur
keytool -keystore server.truststore.jks -alias CARoot -import -file ca-cert -storepass secret -keypass secret -noprompt


#Importer le CA dans le client
keytool -keystore client.truststore.jks -alias CARoot -import -file ca-cert -storepass secret -keypass secret -noprompt
```

Configurer le listener SSL et les propriétés SSL suivante dans *server.properties*

```
ssl.keystore.location=/home/dthibau/Formations/Kafka/github/solutions/ssl/server.keystore.jks
ssl.keystore.password=secret
ssl.key.password=secret
ssl.truststore.location=/home/dthibau/Formations/Kafka/github/solutions/ssl/server.truststore.jks
ssl.truststore.password=secret
security.inter.broker.protocol=SSL
ssl.endpoint.identification.algorithm=
ssl.client.auth=none
```

Démarrer le cluster kafka et vérifier son bon démarrages

Mettre au point un fichier ***client-ssl.properties*** avec :

```
security.protocol=SSL
ssl.truststore.location=/home/dthibau/Formations/Kafka/github/solutions/ssl/client.truststore.jks
ssl.truststore.password=secret
```

Vérifier la connexion cliente avec par exemple

```
./kafka-console-producer.sh --broker-list localhost:9192 --topic ssl
    --producer.config client-ssl.properties
```

7.4 Mise en place monitoring Prometheus, Grafana

Dans un premier temps, démarré une ***JConsole*** et visualiser les Mbeans des brokers, consommateurs et producteurs

Dans un répertoire de travail ***prometheus***

```
wget https://repo1.maven.org/maven2/io/prometheus/jmx/jmx_prometheus_javaagent/0.6/jmx_prometheus_javaagent-0.6.jar
wget https://raw.githubusercontent.com/prometheus/jmx_exporter/master/example_configs/kafka-2_0_0.yml
```

Modifier le script de démarrage du cluster afin de positionner l'agent Prometheus :

```
KAFKA_OPTS="$KAFKA_OPTS -javaagent:$PWD/jmx_prometheus_javaagent-0.2.0.jar=7071:$PWD/kafka-0-8-2.yml" \
  ./bin/kafka-server-start.sh config/server.properties
```

Attention, Modifier le port pour chaque broker

Redémarrer le cluster et vérifier `http://localhost:7071/metrics`

Récupérer et démarrer un serveur prometheus

```
wget https://github.com/prometheus/prometheus/releases/download/v2.0.0/prometheus-2.0.0.linux-amd64.tar.gz
tar -xzf prometheus-*.tar.gz
cd prometheus-*
cat <<'EOF' > prometheus.yml
global:
 scrape_interval: 10s
 evaluation_interval: 10s
scrape_configs:
 - job_name: 'kafka'
   static_configs:
    - targets:
      - localhost:7071
      - localhost:7072
      - localhost:7073
      - localhost:7074
EOF
./prometheus
```

Récupérer et démarrer un serveur Grafana

```
sudo apt-get install -y adduser libfontconfig1
wget https://dl.grafana.com/oss/release/grafana_7.4.3_amd64.deb
```

```
sudo dpkg -i grafana_7.4.3_amd64.deb
```

```
sudo /bin/systemctl start grafana-server
```

Accéder à http://localhost:3000 avec ***admin/admin***

Déclarer la datasource Prometheus

Importer le tableau de bord : https://grafana.com/grafana/dashboards/721

Les métriques des brokers devraient s’afficher