# 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

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

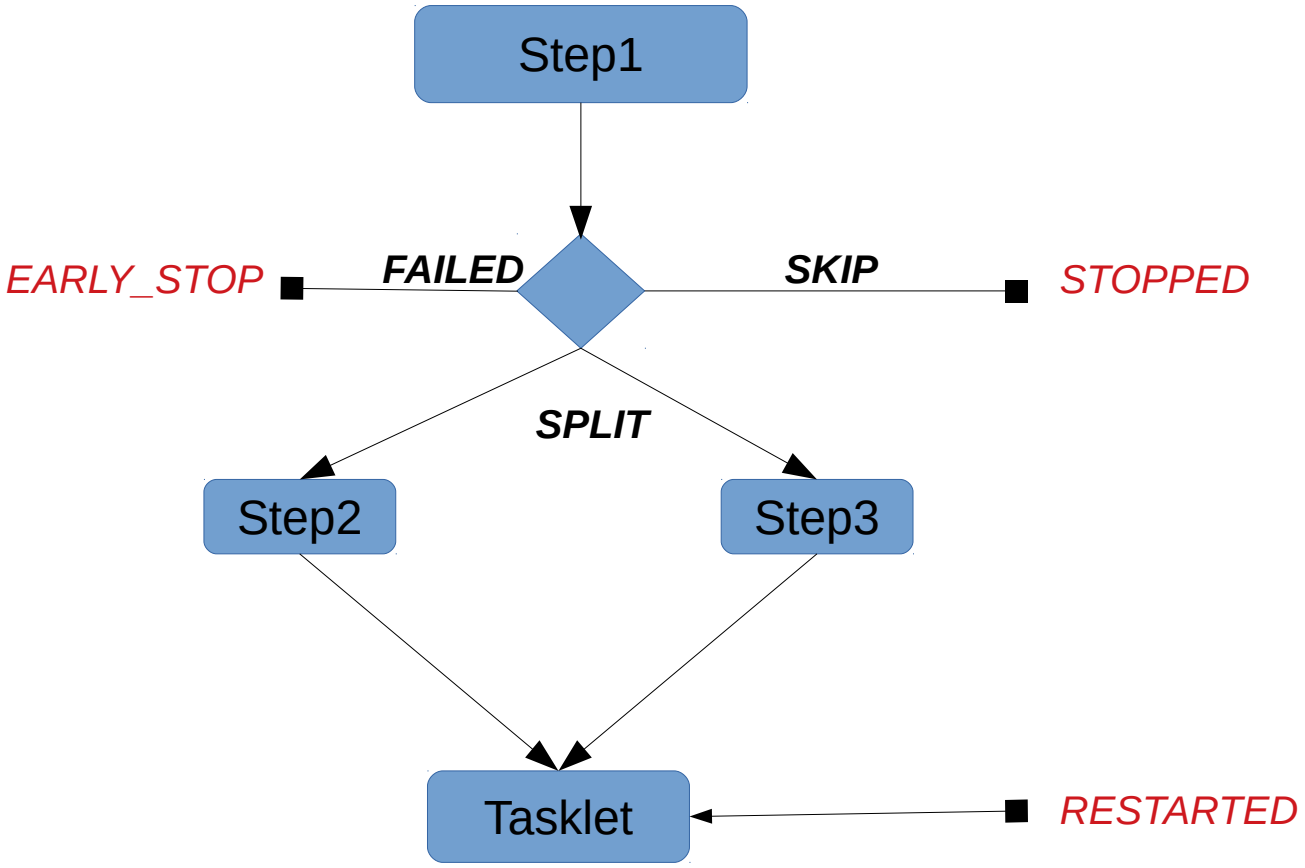
## 2. Listeners et Tasklet

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)

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