

Exercices Spring Batch

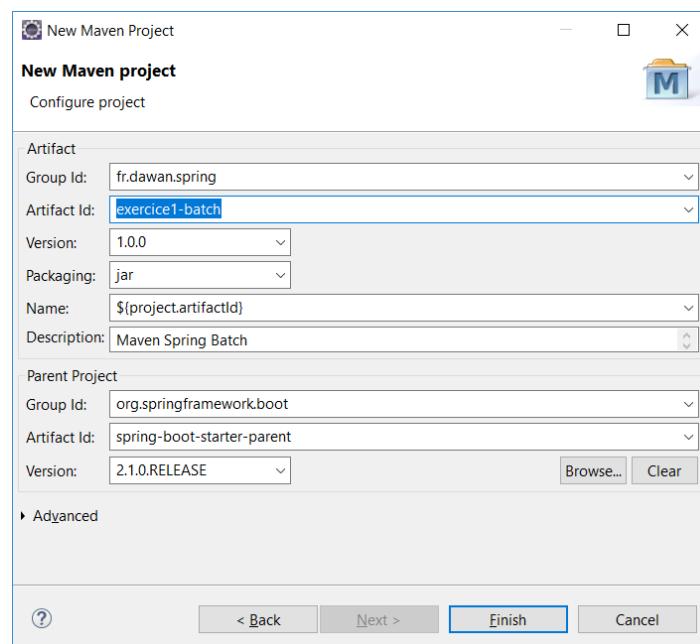
Avec l'IDE de votre choix, il est demandé de créer un workspace `WorkspaceSpring` afin de réaliser les exercices.

Exercice1

Spring dispose de son propre framework de gestion de job en batch. Il permet aussi de faire de la planification de tâche à faire. Le but de cet exercice est d'utiliser les annotations de base de Spring Batch avec Spring Boot avec une configuration basée sur une classe annotée.

Etape 1 : création du projet

Nous allons créer un projet Maven nommé `exercice1-batch`.



L'idée de ce 1^{er} exercice est de lancer une exécution avec spring batch qui permette :

- L'exécution d'un job constitué d'un step

Ajoutez les dépendances Spring

```
<!-- Compile -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-batch</artifactId>
</dependency>

<!-- Test -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

Ajouter un plugin d'exécution

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
```

Etape 2 : création d'une classe exemple

Il est demandé de faire une classe `SampleBatchApplication`

- Injecter un `JobBuilderFactory` pour la construction des Job
- Injecter un `StepBuilderFactory` pour la construction des steps d'un job
- Définir un bean de type `Tasklet` notifiant la fin du job
- Définir un bean de type `Job` pour démarrer le job
- Définir un bean de type `Step` comme élément du job
- Construire une méthode principale

```
package exercicel;

...
public class SampleBatchApplication {
    ...
}
```

Ajouter un fichier de configuration de logback dans le répertoire `src/main/resources`.

Etape 3 : création d'une classe de test

Il est demandé de faire une classe `SampleBatchApplicationTests`

- Définir une méthode de test `testDefaultSettings()` pour valider que la classe principale se termine correctement

```
package exercicel;

public class SampleBatchApplicationTests {
    ...
}
```

Exercice2

Le but de cet exercice est de modifier un fichier texte en un fichier XML. Pour ce faire, il faudra dans cet exercice développer plusieurs classes pour décomposer le traitement batch

Etape 1 : création du projet

Les dépendances ainsi que les plugins sont les mêmes pour la configuration Maven. Il est demandé d'ajouter les dépendances sur Spring oxm, Spring batch. Mettre à jour en fonction de la version souhaitée.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-batch</artifactId>
  <version>2.1.8.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-oxm</artifactId>
  <version>5.1.3.RELEASE</version>
</dependency>
```

Etape 2 : préparation des données et du POJO pour la mapping

Nous voulons lire un fichier `src/main/resources/ExamResult.txt` et écrire un fichier `project/xml/examResult.xml`. Ci-dessous le fichier d'entrée avec des champs séparés par '|' dont les données seront converties au format XML.

Ode Javel		01/02/1985		76
Gaspard Tchao		01/02/1983		39
Jacques Sentu		01/02/1970		61
Marie Golotte		01/02/2002		59
Maude Zarella		01/02/1993		92
Cecile OurKessa		01/02/1965		83

Et une classe POJO, nommée `ExamResult` dont les attributs correspondent aux colonnes du fichier d'entrée.

```
package exercice2;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
import org.joda.time.LocalDate;

@XmlRootElement(...)
public class ExamResult {
    ...
}
```

Nous utilisons des annotations JAXB pour faire le mapping avec des fichiers XML. Il est essentiel d'annoter les getters et non les attributs. A cause de l'usage des dates nous devons développer un adaptateur de format, c'est-à-dire une classe nommée `LocalDateAdapter` qui hérite `XmlAdapter`

et permet de transformer une String en LocalDate.

```
package exercice2;

import javax.xml.bind.annotation.adapters.XmlAdapter;

public class LocalDateAdapter extends XmlAdapter<String, LocalDate>{

    ...
}
```

Etape 3 : faire le mapping de chaque champ

Il est demandé de construire une classe nommée `ExamResultFieldSetMapper` qui implémente `FieldSetMapper` est responsable de l'association de chacun des champs de l'entrée texte sur un attribut d'objet de la classe `ExamResult`. Cette classe doit implémenter une méthode `mapFieldSet` qui retourne une instance d'`ExamResult`.

```
package exercice2;

import org.springframework.batch.item.file.mapping.FieldSetMapper;
import org.springframework.batch.item.file.transform.FieldSet;
import org.springframework.validation.BindException;

public class ExamResultFieldSetMapper implements
FieldSetMapper<ExamResult>{

    ...
}
```

N.B. le reader sera implémenter par la classe `FlatFileItemReader`.

Etape 4 : implémenter la logique métier

Ajouter une classe `ItemProcessor` pour le contrôle sur les données créées. Elle intervient après avoir lu une donnée d'entrée qui doit être transformée avant d'être écrite sur une sortie. C'est dans cette classe qu'il y a de la logique métier. Dans cet exercice, on souhaite filtrer toutes les données dont le resultat est inférieure à 60.

```
package exercice2;

import org.springframework.batch.item.ItemProcessor;

public class ExamResultItemProcessor implements ItemProcessor<ExamResult,
ExamResult>{

    ...
}
```

N.B le writer sera implémenté par la classe `StaxEventItemWriter`.

Etape 5 : construire une observation du traitement

Un `JobExecutionListener` est aussi une classe optionnelle mais elle fournit la possibilité d'effectuer un traitement avant le lancement d'une opération de lecture de fichier txt ou après que le job ait terminé. Par exemple initialiser des variable d'environnement peut être fait avant le début

du job et nettoyer ces variables peut être fait ensuite.

Dans cet exercice, nous souhaitons mesurer le temps passer pour effectuer le traitement d'un job. Pour ce faire, il faut implémenter les méthodes `beforeJob` et `afterJob` pour effectuer 2 mesures de temps et ainsi pouvoir calculer une différence.

De plus, en fonction de son état signaler un éventuel problème (`BatchStatus.FAILED`).

```
package exercice2;

import java.util.List;
import org.joda.time.DateTime;
import org.springframework.batch.core.BatchStatus;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.JobExecutionListener;

public class ExamResultJobListener implements JobExecutionListener{

    ...
}
```

Etape 4 : faire une configuration du job

2 solutions sont possibles au choix

4.1 un contexte XML, dans ce cas il faut installer dans le catalogue de schema d'Eclipse le schema de Spring-Batch.

- la location du schéma : <https://www.springframework.org/schema/batch/spring-batch-4.0.xsd>
- le type de clé: Namespace name
- la key: <https://www.springframework.org/schema/batch/spring-batch-4.0.xsd>

Dans `src/main/resource/spring-batch-context.xml`

Pour configure le job on utilise `FlatFileItemRead` pour lire les éléments du fichier

4.2 une classe de configuration (solution conseillée aujourd'hui), nommée `BatchConfiguration`. Cette classe doit offrir la définir des beans suivants :

- un reader pour notre fichier texte
- un itemProcessor pour le traitement métier
- un writer pour effectuer l'écriture du fichier XML
- un listener pour faire les mesures de temps
- un job pour traiter les fichiers d'entrée
- un step qui comprend le triplet `<reader, itemProcessor, writer>`

L'exercice précédent a montré que des bean factory pouvaient être injectés par Spring automatiquement pour les `JobFactory` et `StepFactory`

Etape 5 : faire l'application principale pour lancer le job

Ajouter un fichier de configuration de logback dans le répertoire `src/main/resources`.

```
package exercise2;

import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessi
ng;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

public class AppMain {
    @SuppressWarnings("resource")
    public static void main(String args[]){
        ...
    }
}
```

Cette classe sera annotée `@SpringApplication`.

Etape 6: exécution

Il ne reste plus qu'à faire une exécution de cette application. Il suffit de suivre la démarche identique de l'exercice précédent. On obtient:

- `mvn clean`
- `mvn install`
- `mvn exec:java -Dexec.mainClass=exercise2.AppMain`

Valider que les fichiers créés en sortie valident la logique métier programmée.

Exercice3

Le but de cet exercice est de lire un fichier CSV pour l'exporter dans une base de données PostgreSQL. Les étapes à suivre sont les suivantes :

- Créer un projet Spring Boot
- Configurer les propriétés de l'application
- Créer un modèle de données
- Créer un DAO
- Créer un step: Reader, Processor, Writer, Listener
- Créer une configuration Batch
- Créer un Web Controller
- Exécuter l'application Spring Boot et d'analyser le résultat
- Ajouter la tolérance et le comptage d'anomalies

Etape 1 : création du projet

Créer un projet Spring Boot avec Maven avec les dépendances adéquates en fonction du SGBD utilisé et de l'ORM employé.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-batch</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Ajouter le plugin spring-boot comme dans les exercices précédents.

Etape2 : configurer les propriétés de l'application

Dans le répertoire `src/main/resources`, définir un fichier `application.properties`. Cette configuration porte sur la connexion à la base de données avec la définition d'un datasource (url, username, password, platform) et la prise en compte d'un schéma de base de

données (`initialize-schema`) mais aussi la non répétition des jobs en échec (`job.enabled`).

N.B. voir <https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>

Définir un fichier `schema-postgresql.sql`, afin de créer un table `student` dans un base `exercice3` avec 3 colonnes (si elle n'existe pas déjà:

- Un `id` de type `Bigserial` comme clé primaire
- Une colonne `first_name` de type `varchar(20)`
- Une colonne `last_name` de type `varchar(20)`
- Une colonne `score` de type `int`

Définir un fichier exemple `student-data.csv`.

```
0,Lara,Leuze,10/10/1990,61
1,Jean,Transene,8/10/1991,74
2,Ella,Stick,21/11/1988,55
3,Mehdi,Moitout,8/8/1998,91
4,Jade,Orlefrites,11/6/1992,68
5,Laurent,Outant,4/2/1994,91
```

Etape3 : création d'un data modèle

Dans le répertoire `src/main/java`, définir un package `exercice3.model`. Ajouter une classe modèle nommée `Student`, considérée comme un POJO elle possède les attributs qui vous intéressent depuis le fichier d'entrée : `id`, `firstname`, `lastname`, `score`. Il sera important d'y ajouter une méthode `toString()` pour afficher leur contenu en cas de mise au point.

Etape4 : création d'un DAO

Dans le répertoire `src/main/java`, définir un package `exercice3.dao`.

Créer une interface `StudentDao`, dans cet exercice nous n'utilisons pas `spring-data`. Elle comprend 2 obligations de codage :

- Insérer une liste de `Student`
- Lire tous les `Students`

Créer un package `exercice3.dao.impl` et une implémentation de ce DAO, nommée `StudentDaoImpl`, qui hérite de `JdbcDaoSupport` et implémente le contrat précédent `StudentDao`. Cette classe est annotée `@Repository` et Spring injecte la définition d'un `DataSource`. Cette classe possède une méthode annotée `@PostConstruct` pour conserver une référence au `DataSource`, une définition pour l'insertion et la lecture de `Student`.

Pour l'insertion, la méthode contient une requête JDBC spécifique et l'emploi d'un `PreparedStatement`. L'initialisation de celui-ci se fait par la définition d'une instance de `BatchPreparedStatementSetter`, et la redéfinition des méthodes :

- `setValues()` : permet d'initialiser le `PreparedStatement`

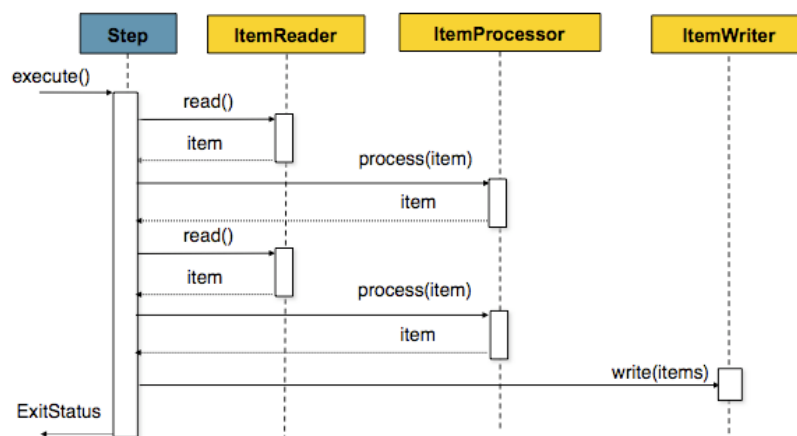
- `getBatchSize()` : permet de terminer la boucle sur la liste de `Student`.

L'exécution de la requête est effectuée par l'emploi d'une instance de `JdbcTemplate` et sa méthode `batchUpdate()`.

Pour la lecture de tous les `Students`, le procédé est le même : une requête `Jdbc` spécifique, l'usage d'une instance de `JdbcTemplate` et de sa méthode `queryForList`. A partir du résultat, il faut créer une liste de `Student`. Attention la liste reçue est une liste de `Map` c'est-à-dire de clé, valeur qu'il convient de transformer en objet.

Etape5 : création d'une étape de travail: Reader, Processeur, Writer, Listener

Dans un package `exercice3.step`, il est demandé de créer les classes suivantes pour implémenter un chunk:



- Une classe `StudentReader` qui hérite de `FlatFileItemReader<Student>` pour la lecture du fichier CSV en `Student`. Le constructeur de cette classe contient toute la configuration basée sur un path en paramètre,
- Une classe `StudentProcessor` implémentant `ItemProcessor` qui affiche les `Students` avec une méthode `process` qui étant donné un `Student` retourne un `Student` si son score est supérieur à 50
- Une classe `StudentWriter` qui hérite de `ItemWriter<Student>` pour insérer les données via l'utilisation du `StudentDao`. Elle possède une méthode `write` ayant en paramètre la liste des `Students`,
- Une classe `StudentListener` qui hérite de `JobExecutionListenerSupport` afin de contrôler qu'après le job la table à évoluer en utilisant le DAO. Il faut donc implémenter `afterJob` et utiliser un `StudentDao`.

Etape6 : création d'une classe de Configuration

Dans le répertoire `src\main\java`, définir un package `exercice3.config`. Créer une classe `BatchConfiguration`. Cette classe doit être annotée `@Configuration` et `active Batch Processing`. Qui expose les beans essentiels pour le lancement du job (voir exercice2) : job, step et listener

```
package exercice3.config;
```

```
...
public class BatchConfig {
    ...
}
```

Le paramètre d'entrée pour la méthode `chunk` de `StepBuilder` spécifie le nombre d'éléments à lire avant l'écriture via `ItemWriter`.

Etape7 : création d'un WebController

Dans le répertoire `src\main\java`, définir un package `exercice3.controller`.

Créer une classe `WebController`, qui soit un service REST pour exposer derrière l'uri `/runjob` le lancement du Job et utiliser la classe `JobLauncher`. Pour cela la classe doit supporter les annotations : `@RestController`, `@EnableAutoConfiguration`. De plus comme cette classe est définie dans un package, il faut préciser les packages à scanner pour trouver les autres composants.

Le contexte Spring va injecter un `JobLauncher`, ainsi que le Job défini dans la configuration, grâce à la déclaration de 2 attributs de même type.

La classe `WebController`, comprend aussi une méthode nommée `handleGet()` qui retourne une `String` et qui est l'opération du Web Service REST. Pour cela cette méthode est annotée `@RequestMapping`. La définition de la méthode crée un `JobParameter` pour le temps du départ et effectue le lancement via l'appel de la méthode `run()` sur l'instance de `JobLauncher`.

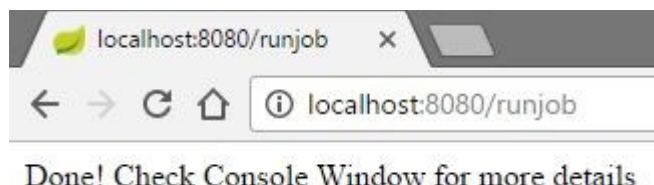
Ajouter dans cette classe une méthode principale afin de lancer le service pour qu'il soit invocable depuis un navigateur.

Etape8 : Fabrication d'un livrable

Lancer la commande maven : `mvn clean install`

Executer avec maven et son plugin spring-boot : `mvn spring-boot :run`

Lancer un navigateur avec l'url : `http://localhost:8080/runjob`



Ouvrez `phAdminIV` pour vérifier la table `student` dans la base de données `exercice3`.

Etape9 : Modification du mapping facultatif

Il est demandé de prendre en compte les dates de naissance et le pourcentage de réussite comme

dans l'exercice précédent. Mettre à jour le schéma de table (ALTER TABLE student ADD birth date). De plus, il faut importer la classe de transformation de String en LocalDate et réciproquement. Enfin le Mapper est à mettre à jour.

Le processor devra filtrer les students ayant moins de 50 de taux de réussite.

Afin de ne pas créer de conflit il est utile d'appliquer un TRUNCATE sur la table afin de la purger.

Etape10 : Anomalie de données

Modifier La définition du step afin d'introduire la possibilité de tolérance aux pannes. Ajouter une fonction pour la définition d'une politique d'éviction. Dans ce but, il est demandé :

- Une classe `CSVFileVerification` qui implémente `SkipPolicy` pour la gestion des problèmes de fichier non trouvé. Elle possède une méthode `shouldSkip` qui retourne un booléen qui en cas de false entraine l'arrêt du step.

La fonction `skipLimit` prend en paramètre la limite d'erreur à ne pas atteindre. Puis utiliser la méthode `skipPolicy` avec une instance de la classe précédente.

Etape11 : Test Spring test

Après avoir ajouté la dépendance `starter-test`, il est demandé d'écrire dans le répertoire de test de Eclipse un test unitaire pour contrôler que le fichier contient des données utilisables pour en faire des Student.

Exécuter les tests avec la phase de test de maven.

Exercice4

Le but de cet exercice est d'utiliser l'approche parallèle du scheduling avec Spring Batch. Dans Spring Batch, le "Partitioning" crée "plusieurs threads pour traiter un ensemble de données ". Par exemple, supposons que vous avez 100 fichiers CSV, nous souhaitons traiter les fichiers en même temps.

Normalement, avec un processus, la séquence commence de 1 à 100, un exemple de thread unique. Le processus est estimé à 10 minutes pour terminer.

Avec le "Partitioning", nous pouvons démarrer 10 threads pour traiter 10 fichiers chacun basé sur les noms uniques. Maintenant, le processus peut prendre seulement 1 minute pour terminer.

Pour implémenter la technique de « partitionnement », vous devez comprendre la structure des données d'entrée à traiter, afin de pouvoir planifier correctement les dépendances.

Créer un "Partitioner" job, qui a 10 threads, chaque thread va lire les enregistrements de la base de données, en fonction d'un intervalle d' «id».

Dans la base PostgreSQL, il est demandé de créer un grand nombre de Students (> 20)

Tout d'abord, créer une implémentation de Partitioner, et indiquer l' intervalle de partitionnement dans le Contexte d'exécution. Plus tard, vous déclarerez la même chose à partir d'Id dans la configuration.

Dans ce cas, la plage de partitionnement peut être la suivante :

Thread 1 = 1 - 10

Thread 2 = 11 - 20

Thread 3 = 21 - 30

.....

Thread 10 = 91 – 100

```
package exercice4;
import java.util.HashMap;
import java.util.Map;
import org.springframework.batch.core.partition.support.Partitioner;
import org.springframework.batch.item.ExecutionContext;

public class RangePartitioner implements Partitioner {

    @Override
    public Map<String, ExecutionContext> partition(int gridSize) {

        Map<String, ExecutionContext> result
            = new HashMap<String, ExecutionContext>();

        int range = 10;
        int fromId = 1;
        int toId = range;
        for (int i = 1; i <= gridSize; i++) {
            ExecutionContext value = new ExecutionContext();
            System.out.println("\nStarting : Thread" + i);
```

```

        System.out.println("fromId : " + fromId);
        System.out.println("toId : " + toId);

        value.putInt("fromId", fromId);
        value.putInt("toId", toId);

        // fournir un nom au thread
        value.putString("name", "Thread" + i);

        result.put("partition" + i, value);
        fromId = toId + 1;
        toId += range;
    }

    return result;
}
}

```

Passez en revue le fichier la configuration Spring Batch:

Pour le partitionneur, la taille de la grille et le nombre de threads.

Pour le haritube pagingItemReader, un exemple de lecteur jdbc, les valeurs de 'stepExecutionContext[fromId, toId]' seront injectées par le ExecutionContext dans rangePartitioner.

Pour le haritusien itemProcessor, les valeurs de 'stepExecutionContext[nom]' seront injectées par le ExecutionContext dans rangePartitioner.

Pour les auteurs, chaque thread produira les enregistrements dans un fichier csv différent, avec le format de nom de fichier - users.processed[fromId] -[toId].csv.