

# Spring Batch

**Achref El Mouelhi**

Docteur de l'université d'Aix-Marseille  
Chercheur en programmation par contrainte (IA)  
Ingénieur en génie logiciel

`elmouelhi.achref@gmail.com`



spring

# Plan

- 1 Introduction
- 2 Exemple avec un Job et un Step
- 3 Exemple avec un Job, un Tasklet et un Step

# Spring Batch

## Spring Batch

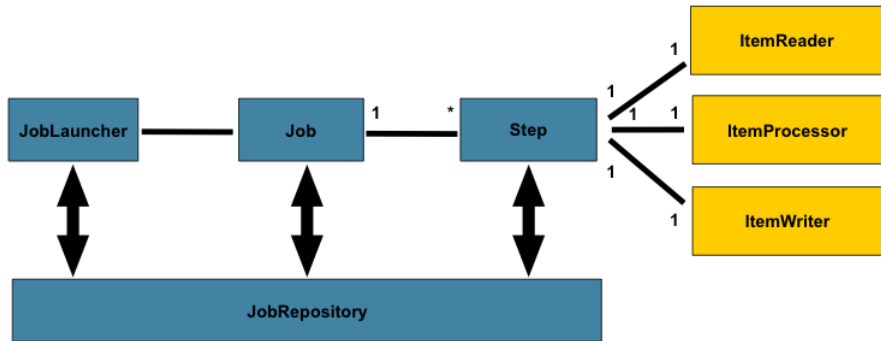
- Framework open-source (unique) pour le traitement par lots
- Fournissant des fonctions réutilisables pour des grands volumes de données
- Fonctionnement principal = { lecture de données  $\Rightarrow$  traitement de données  $\Rightarrow$  écriture de données }
- Configurable par **XML** ou classes **Java**

# Spring Batch

## Exemple réel : opération de pointage

- Les employés d'une entreprise pointent en début et fin de journée
- La pointeuse renvoie généralement un fichier texte (d'extension `.csv`)
- Il faut lire les données, supprimer les pointages doubles, corriger le format de date et tout sauvegarder dans la base de données
- Traitement périodique et répétitif
- Pour automatiser le lancement, on peut utiliser **Spring Batch**

# Spring Batch



**Source :** documentation officielle

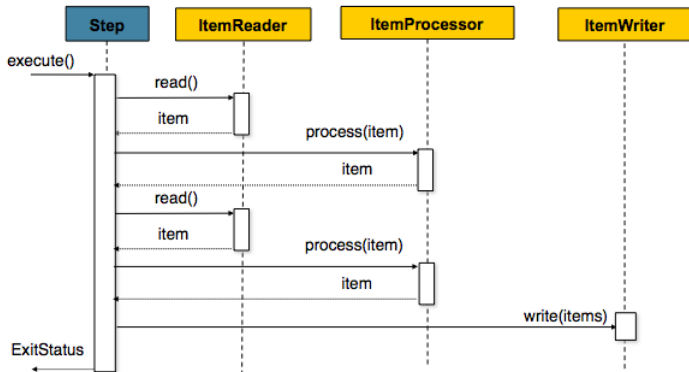
# Spring Batch

## Explication

- **Job** = { Step } (conteneur de Step).
- **Step** = `ItemReader` + `ItemProcessor` + `ItemWriter`.
- **Chunk** indique le nombre d'item à charger en mémoire. Par exemple : si on passe la valeur 4 à `chunk`, on lit les items un par un, une fois que le nombre d'éléments lus est égal à 4, le bloc entier est traité par `ItemProcessor` puis écrit par `ItemWriter`, et on continue avec les autres de la même manière.
- **JobLauncher** exécute les jobs.
- **JobRepository** contient les méta-données relatives aux différents traitements.

# Spring Batch

## Exemple avec chunk (2)



**Source :** documentation officielle

# Spring Batch

## Création de projet Spring Boot

- Aller dans `File > New > Other`
- Chercher `Spring`, dans `Spring Boot` sélectionner `Spring Starter Project` et cliquer sur `Next >`
- Saisir
  - `first-spring-batch` dans `Name`,
  - `com.example` dans `Group`,
  - `firstspringbatch` dans `Artifact`
  - `com.example.demo` dans `Package`
- Cliquer sur `Next`
- Chercher et cocher les cases correspondantes aux `Spring Data JPA`, `MySQL Driver`, `Spring Web`, `Spring Boot DevTools`, `Lombok` et `Spring Batch`
- Cliquer sur `Next` puis sur `Finish`

# Spring Batch

## Explication

- Le package contenant le point d'entrée de notre application (la classe contenant le `public static void main`) est `com.example.demo`
- Tous les autres packages `dao`, `model...` doivent être dans le package `demo`.

© Achref EL ME

# Spring Batch

## Explication

- Le package contenant le point d'entrée de notre application (la classe contenant le `public static void main`) est `com.example.demo`
- Tous les autres packages `dao`, `model...` doivent être dans le package `demo`.

## Pour la suite, nous considérons

- une entité `Personne` à définir dans `com.example.demo.model`
- une interface DAO `PersonneRepository` à définir dans `com.example.demo.dao`
- un contrôleur REST `PersonneController` à définir dans `com.example.demo.dao`

# Spring Batch

Créons une entité `Personne` dans `com.example.demo.model`

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
public class Personne {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long num;
    private String nom;
    private String prenom;
    private Long salaire;
}
```

# Spring Batch

## Préparons notre interface DAO `PersonneRepository`

```
package com.example.demo.dao;

import org.springframework.data.jpa.repository.
    JpaRepository;

import com.example.demo.model.Personne;

public interface PersonneRepository extends
    JpaRepository<Personne, Long> {

}
```

# Spring Batch

**Considérons le fichier `data.csv` à créer dans**  
`src/main/resources`

```
nom_p,prenom_p,salaire_p  
wick,john,1700  
dalton,jack,2000  
maggio,carol,1900  
hoffman,mike,2300
```

# Spring Batch

## Contexte et objectif

- La **RH** de notre entreprise nous fournit régulièrement un fichier `.csv` contenant les noms et prénoms des nouvelles recrues ainsi que les salaires négociés.
- À partir de ce fichier, nous souhaitons ajouter ces personnes dans la base de données.
- Nous souhaitons aussi que le nom et la première lettre du prénom soient écrits en majuscule.

Dans `application.properties`, ajoutons les données permettant la connexion à la base de données et la configuration de `Hibernate`

```
spring.datasource.url = jdbc:mysql://localhost:3306/batch?  
    serverTimezone=UTC  
spring.datasource.username = root  
spring.datasource.password = root  
spring.jpa.hibernate.ddl-auto = update  
spring.jpa.show-sql = true  
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.  
    MySQL8Dialect  
spring.jpa.hibernate.naming.physical-strategy = org.hibernate.boot.  
    model.naming.PhysicalNamingStrategyStandardImpl
```

© Achref EL

Dans `application.properties`, ajoutons les données permettant la connexion à la base de données et la configuration de `Hibernate`

```
spring.datasource.url = jdbc:mysql://localhost:3306/batch?  
    serverTimezone=UTC  
spring.datasource.username = root  
spring.datasource.password = root  
spring.jpa.hibernate.ddl-auto = update  
spring.jpa.show-sql = true  
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.  
    MySQL8Dialect  
spring.jpa.hibernate.naming.physical-strategy = org.hibernate.boot.  
    model.naming.PhysicalNamingStrategyStandardImpl
```

Indiquons également l'emplacement de notre fichier `CSV`

```
inputFile=classpath:/data.csv
```

Dans `application.properties`, ajoutons les données permettant la connexion à la base de données et la configuration de `Hibernate`

```
spring.datasource.url = jdbc:mysql://localhost:3306/batch?
serverTimezone=UTC
spring.datasource.username = root
spring.datasource.password = root
spring.jpa.hibernate.ddl-auto = update
spring.jpa.show-sql = true
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.
MySQL8Dialect
spring.jpa.hibernate.naming.physical-strategy = org.hibernate.boot.
model.naming.PhysicalNamingStrategyStandardImpl
```

Indiquons également l'emplacement de notre fichier `CSV`

```
inputFile=classpath:/data.csv
```

Demandons à `Spring Boot` de ne pas lancer les jobs au démarrage de l'application

```
spring.batch.job.enabled = false
```

Dans `application.properties`, ajoutons les données permettant la connexion à la base de données et la configuration de `Hibernate`

```
spring.datasource.url = jdbc:mysql://localhost:3306/batch?
serverTimezone=UTC
spring.datasource.username = root
spring.datasource.password = root
spring.jpa.hibernate.ddl-auto = update
spring.jpa.show-sql = true
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.
MySQL8Dialect
spring.jpa.hibernate.naming.physical-strategy = org.hibernate.boot.
model.naming.PhysicalNamingStrategyStandardImpl
```

Indiquons également l'emplacement de notre fichier `CSV`

```
inputFile=classpath:/data.csv
```

Demandons à `Spring Boot` de ne pas lancer les jobs au démarrage de l'application

```
spring.batch.job.enabled = false
```

Autorisons `Spring Batch` à générer ses tables nécessaires pour la gestion de lots

```
spring.batch.initialize-schema=always
```

# Spring Batch

## Créons une classe de configuration pour Spring Batch

```
package com.example.demo.batch;

import org.springframework.batch.core.configuration.
    annotation.EnableBatchProcessing;
import org.springframework.context.annotation.
    Configuration;

@Configuration
@EnableBatchProcessing
public class SpringBatchConfig {

}
```

# Spring Batch

Dans la classe `SpringBatchConfig`, nous commençons par injecter les 5 éléments indispensables pour le lancement d'un job

```
@Configuration
@EnableBatchProcessing
public class SpringBatchConfig {
    @Autowired private JobBuilderFactory jobBuilderFactory;
    @Autowired private StepBuilderFactory stepBuilderFactory;
    @Autowired private ItemReader<Personne> itemReader;
    @Autowired private ItemWriter<Personne> itemWriter;
    @Autowired private ItemProcessor<Personne, Personne> itemProcessor;
}
```

© Achre

# Spring Batch

Dans la classe `SpringBatchConfig`, nous commençons par injecter les 5 éléments indispensable pour le lancement d'un job

```
@Configuration
@EnableBatchProcessing
public class SpringBatchConfig {
    @Autowired private JobBuilderFactory jobBuilderFactory;
    @Autowired private StepBuilderFactory stepBuilderFactory;
    @Autowired private ItemReader<Personne> itemReader;
    @Autowired private ItemWriter<Personne> itemWriter;
    @Autowired private ItemProcessor<Personne, Personne> itemProcessor;
}
```

## Les imports nécessaires

```
import org.springframework.batch.core.configuration.annotation.
    JobBuilderFactory;
import org.springframework.batch.core.configuration.annotation.
    StepBuilderFactory;
import org.springframework.batch.item.ItemProcessor;
import org.springframework.batch.item.ItemReader;
import org.springframework.batch.item.ItemWriter;
```

# Spring Batch

Définissons un Bean pour le Job (dans SpringBatchConfig)

```
@Bean
public Job job(Step step1) {
    return jobBuilderFactory.get("Recrutement") // le nom de notre job
        .start(step1)
        .build();
}
```

© Achref EL MOUEZ

# Spring Batch

Définissons un Bean pour le Job (dans SpringBatchConfig)

```
@Bean
public Job job(Step step1) {
    return jobBuilderFactory.get("Recrutement") // le nom de notre job
        .start(step1)
        .build();
}
```

Nous devons donner à Spring plus de détail sur le Step

```
@Bean
public Step step1() {
    return stepBuilderFactory.get("première étape: chargement de fichier"
        )
        .<Personne, Personne>chunk(2)
        .reader(itemReader)
        .processor(itemProcessor)
        .writer(itemWriter)
        .build();
}
```

## Définissons un Bean pour la lecture de données (dans SpringBatchConfig)

```
@Bean
public FlatFileItemReader<Personne> reader(@Value("${inputFile}")
    Resource resource) {
    return new FlatFileItemReaderBuilder<Personne>()
        .name("personItemReader")
        .resource(resource)
        .linesToSkip(1)
        .delimited()
        .names(new String[]{"nom", "prenom", "salaire"})
        .fieldSetMapper(new BeanWrapperFieldSetMapper<Personne>() {{
            setTargetType(Personne.class);
        }})
        .build();
}
```

## Définissons un Bean pour la lecture de données (dans SpringBatchConfig)

```
@Bean
public FlatFileItemReader<Personne> reader(@Value("${inputFile}")
    Resource resource) {
    return new FlatFileItemReaderBuilder<Personne>()
        .name("personItemReader")
        .resource(resource)
        .linesToSkip(1)
        .delimited()
        .names(new String[]{"nom", "prenom", "salaire"})
        .fieldSetMapper(new BeanWrapperFieldSetMapper<Personne>() {{
            setTargetType(Personne.class);
        }})
        .build();
}
```

## Les imports nécessaires

```
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.builder.
    FlatFileItemReaderBuilder;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.io.Resource;
```

# Spring Batch

## Explication

- `@Value("${inputFile}") Resource resource` : permet de récupérer le nom de fichier défini dans `application.properties` et l'injecter dans `resource`
- `linesToSkip(1)` : pour ignorer la première ligne (contenant l'entête) de notre fichier CSV
- `delimited()` : pour construire un objet `DelimitedLineTokenizer`
- `names()` : pour spécifier les noms des attributs de la classe `Personne` présents dans le fichier
- `fieldSetMapper()` : pour spécifier le type de mapping (ici le type de sortie est un objet de la classe `Personne`)

Définissons un `Component` appelé `PersonneItemProcessor` pour spécifier ce qu'on fera avec l'objet lu

```
package com.example.demo.batch;

import org.springframework.batch.item.ItemProcessor;
import org.springframework.stereotype.Component;
import com.example.demo.model.Personne;

@Component
public class PersonneItemProcessor implements ItemProcessor<Personne,
    Personne> {

    @Override
    public Personne process(final Personne personne) throws
        Exception {
        var nom = personne.getNom().toUpperCase();
        personne.setNom(nom);
        var prenom = personne.getPrenom();
        prenom = prenom.substring(0, 1).toUpperCase() + prenom.
            substring(1).toLowerCase();
        personne.setPrenom(prenom);
        return personne;
    }
}
```

## Définissons un `Component` appelé `PersonneItemWriter` pour écrire l'objet transformé

```
package com.example.demo.batch;

import java.util.List;

import org.springframework.batch.item.ItemWriter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import com.example.demo.dao.PersonneRepository;
import com.example.demo.model.Personne;

@Component
public class PersonneItemWriter implements ItemWriter<Personne> {
    @Autowired
    private PersonneRepository personneRepository;

    @Override
    public void write(List<? extends Personne> items) throws
        Exception {
        System.out.println(items);
        personneRepository.saveAll(items);
    }
}
```

# Spring Batch

Créons un contrôleur (qui nous permettra de lancer le Job) et commençons par injecter le Job **et** JobLauncher

```
package com.example.demo.controller;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class JobInvokerController {

    @Autowired
    private JobLauncher jobLauncher;

    @Autowired
    private Job processJob;

}
```

## Ajoutons l'action qui permettra de lancer le Job

```
package com.example.demo.controller;

@RestController
public class JobInvokerController {

    @Autowired
    private JobLauncher jobLauncher;

    @Autowired
    private Job processJob;

    @RequestMapping("/loadData")
    public String handle() throws Exception {

        JobParameters jobParameters = new JobParametersBuilder()
            .addLong("time", System.currentTimeMillis())
            .toJobParameters();
        jobLauncher.run(processJob, jobParameters);

        return "Le job a été exécuté";
    }
}
```

# Spring Batch

## Objectif

- Supprimer le fichier `data.csv` après ajout dans la base de données
- Utiliser un `Tasklet` après le premier `Step` pour supprimer le fichier

# Spring Batch

## Tasklet

- Comme un Step
  - sans `ItemWriter`
  - généralement sans valeur de retour
- Utilisé souvent pour
  - appeler une procédure stockée
  - exécuter un script

# Spring Batch

Dans la classe `SpringBatchConfig`, ajoutons la déclaration de notre `Tasklet`

```
@Configuration
@EnableBatchProcessing
public class SpringBatchConfig {
    @Autowired private JobBuilderFactory jobBuilderFactory;
    @Autowired private StepBuilderFactory stepBuilderFactory;
    @Autowired private ItemReader<Personne> itemReader;
    @Autowired private ItemWriter<Personne> itemWriter;
    @Autowired private ItemProcessor<Personne, Personne> itemProcessor;
    @Autowired private Tasklet tasklet;

    // ...
}
```

# Spring Batch

Dans la classe `SpringBatchConfig`, ajoutons la déclaration de notre `Tasklet`

```
@Configuration
@EnableBatchProcessing
public class SpringBatchConfig {
    @Autowired private JobBuilderFactory jobBuilderFactory;
    @Autowired private StepBuilderFactory stepBuilderFactory;
    @Autowired private ItemReader<Personne> itemReader;
    @Autowired private ItemWriter<Personne> itemWriter;
    @Autowired private ItemProcessor<Personne, Personne> itemProcessor;
    @Autowired private Tasklet tasklet;

    // ...
}
```

L'import nécessaire

```
import org.springframework.batch.core.step.tasklet.Tasklet;
```

# Spring Batch

Modifions le Job précédent pour lancer le Tasklet après step1

```
@Bean
public Job job(@Qualifier("step1") Step step1, @Qualifier("deleteFile")
    Step deleteFile) {
    return jobBuilderFactory.get("Recrutement")
        .start(step1)
        .next(deleteFile)
        .build();
}
```

© Achref EL

# Spring Batch

Modifions le Job précédent pour lancer le Tasklet après step1

```
@Bean
public Job job(@Qualifier("step1") Step step1, @Qualifier("deleteFile")
    Step deleteFile) {
    return jobBuilderFactory.get("Recrutement")
        .start(step1)
        .next(deleteFile)
        .build();
}
```

Définissons le Tasklet défini dans le Job

```
@Bean
public Step deleteFile() {
    return this.stepBuilderFactory.get("deuxième étape : suppression
        du fichier CSV")
        .tasklet(tasklet)
        .build();
}
```

Définissons un `Component` appelé `FileDeletingTasklet` pour la suppression de notre fichier CSV

```

package com.example.demo.batch;

import java.io.File;

import org.springframework.batch.core.StepContribution;
import org.springframework.batch.core.UnexpectedJobExecutionException;
import org.springframework.batch.core.scope.context.ChunkContext;
import org.springframework.batch.core.step.tasklet.Tasklet;
import org.springframework.batch.repeat.RepeatStatus;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.io.Resource;
import org.springframework.stereotype.Component;

@Component
public class FileDeletingTasklet implements Tasklet {

    @Value("${inputFile}")
    private Resource resource;

    public RepeatStatus execute(StepContribution contribution, ChunkContext chunkContext)
        throws Exception {
        File file = resource.getFile();
        boolean deleted = file.delete();
        if (!deleted) {
            throw new UnexpectedJobExecutionException("Impossible de supprimer le
                fichier " + file.getPath());
        }
        System.out.println("Fichier supprimé : " + file.getPath());
        return RepeatStatus.FINISHED;
    }
}

```

# Spring Batch

## Rien à changer dans le contrôleur précédent

```
@RestController
public class JobInvokerController {

    @Autowired
    private JobLauncher jobLauncher;

    @Autowired
    private Job processJob;

    @RequestMapping("/loadData")
    public String handle() throws Exception {

        JobParameters jobParameters = new JobParametersBuilder()
            .addLong("time", System.currentTimeMillis())
            .toJobParameters();
        jobLauncher.run(processJob, jobParameters);

        return "Le job a été exécuté";
    }
}
```

# Spring Batch

**Lancez le projet et allez à `localhost:8080/loadData` et vérifiez la présence des messages suivants**

```
....  
Executing step: [deuxième étape : suppression du fichier CSV]  
Fichier supprimé : C:\Users\elmou\eclipse-workspace\first-spring-batch\  
target\classes\data.csv  
Step: [deuxième étape : suppression du fichier CSV] executed in 368ms  
Job: [SimpleJob: [name=Recrutement]] completed with the following  
parameters: [{time=1600800074441}] and the following status: [COMPLETED] in 2s114ms  
....
```

# Spring Batch

## Explication

- Un message nous indique que le fichier `data.csv` a été supprimé
- Ce n'est pas le `data.csv` qu'on a créé dans `src/main/resources`
- Le chemin affiché indique que le fichier supprimé se trouvait dans `first-spring-batch/target/classes`
- En effet, **Spring** a déplacé `data.csv` qu'on a créé dans `src/main/resources` dans `target/classes/`
- Faites un clic droit sur le projet et cliquez sur **Refresh** et allez vérifiez que **Spring** a de nouveau déplacé `data.csv` dans `first-spring-batch/target/classes`

# Spring Batch

## Question

Pourquoi **Spring Boot** a t-il redémarré l'application après suppression de fichier ?

© Achref EL MOU

# Spring Batch

## Question

Pourquoi **Spring Boot** a t-il redémarré l'application après suppression de fichier ?

## Réponse

**Devtools** a détecté un changement (suppression de fichier) et par conséquence a redémarré l'application.