

# Le traitement par lot avec Spring Batch

---

David THIBAU – 2021

[david.thibau@gmail.com](mailto:david.thibau@gmail.com)



# Agenda

---

## Batch processing

- Implémentations de Batch processing
- Spring Batch : Présentation
- Spring Batch : Architecture
- Spring Batch : Concepts

## Spring Batch et Spring Boot

- Rappels Spring Boot : Auto-configuration et annotations
- Dépendances Spring Batch
- Auto-configuration, Surcharge de configuration

## Configuration Jobs

- Alternatives pour la configuration
- Paramètres de configuration
- Configuration du Repository, du Launcher
- Exécution des Jobs

## Steps

- Chunk et transactions
- Tasklet Step
- Séquencement des steps, notion de flow
- Externalisation du flow, binding des attributs de job et de steps

## ItemReader/ItemWrite

- Reader, Writer, Stream
- Fichiers à plats, XML, JSON, BD
- ItemReader/Writer personnalisés
- Item processor, composition de processeurs
- 

## Pour aller + loin

- Scaling et traitement parallèle
- Répétition et ré-essai
- Tests unitaires
- Le projet Spring Batch Integration
- Surveillance et métriques



# Batch Processing

---



# Usage

---

Typiquement, un traitement par lot :

- Lit un grand nombre d'enregistrements à partir d'une base de données, d'un fichier ou d'une file d'attente.
- Traite les données
- Réécrit les données sous une forme modifiée.



---

Spring Batch est une des seules offres open source qui fournit un framework robuste et scalable au niveau de l'entreprise.



# Scénarios métiers

---

- Validez périodiquement le traitement (commit) ou transaction par lot complet
- Traitement parallèle, massivement parallèle
- Traitement séquentiel des étapes dépendantes (avec des extensions aux traitements pilotés par workflow)
- Traitement par étapes et axé sur les messageries d'entreprise
- Reprise sur erreur
- Traitement partiel: ignorer certains enregistrements (par exemple, lors d'une reprise)



# Principes et recommandations

---

Simplifiez autant que possible et évitez de construire des structures logiques complexes en un seul batch.

Gardez le traitement et le stockage des données physiquement proches

Minimisez l'utilisation des ressources système, en particulier les I/O. Effectuez autant d'opérations que possible en mémoire.

Ne pas faire pas deux fois les traitements.

Allouez suffisamment de mémoire dès le départ pour éviter des réallocation en cours du processus.

Envisager le pire concernant l'intégrité des données. Insérez des contrôles adéquats.

Utiliser des checksums pour la validation interne. Par exemple, un fichier à plat peut avoir un enregistrement de fin indiquant le total des enregistrements et un agrégat des champs clés

Planifiez et exécutez des tests de résistance au plus tôt dans un environnement de production avec des volumes de données réalistes.





# Batch Processing

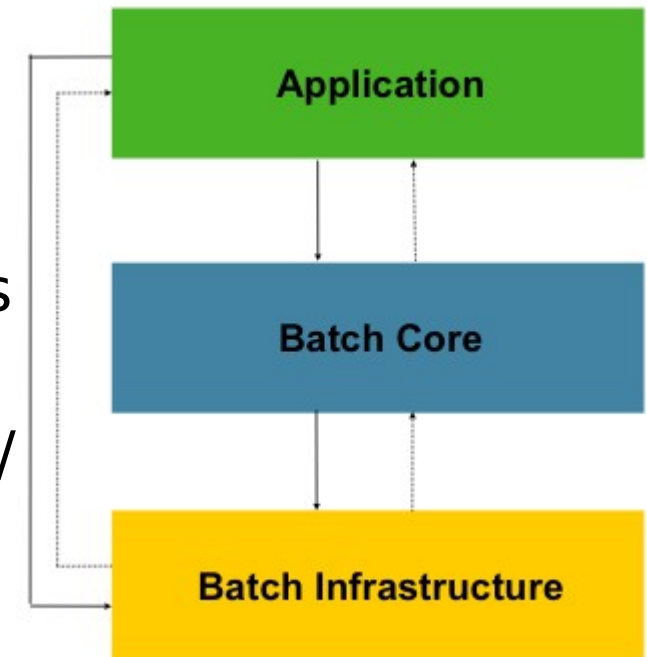
---

## Spring Batch Architecture

# Architecture

Spring Batch propose une architecture en couche :

- L'application contient tous les jobs batch et le code custom fourni par les développeurs
- Le cœur contient les classes d'exécution principales nécessaires pour lancer et contrôler un job.
- L'infrastructure contient des readers/writers et services utilisés par le coeur et par l'application





# Batch Processing

---

Concepts



# Introduction

---

Les concepts généraux du traitement par lots utilisés dans Spring Batch sont classiques.

Il y a :

- Des Jobs
- Constitués d'étapes : les Steps
- Elles-mêmes constitués d'unités de traitement : *ItemReader*, *ItemProcessor* et *ItemWriter*.

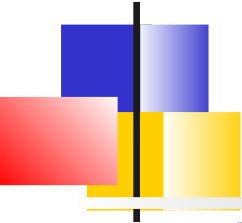


# Apports Spring

---

Spring avec ses patterns (IoC, Template, callback, ...) apporte :

- Des améliorations significative en respect d'une séparation claire des préoccupations (SoC).
- Des couches architecturales clairement délimitées et des services fournis en tant qu'interfaces.
- Des implémentations simples et par défaut qui permettent une adoption rapide et une facilité d'utilisation
- De l'extensibilité



# ItemReader/Writter disponibles

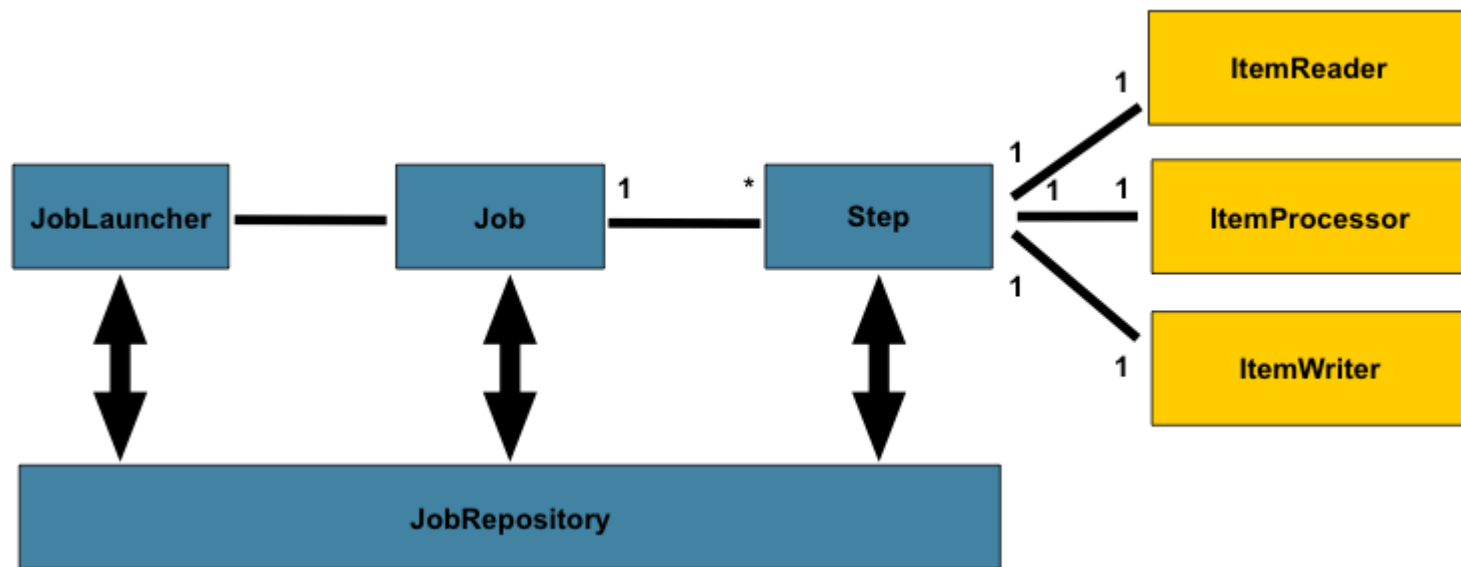
---

FlatFile, StaX XML, JSON

JMS, Amqp, Kafka

Jdbc, Hibernate, Jpa, StoredProcedure,

Mongo, Neo4j





# *Job*

---

Un ***Job*** est une entité qui encapsule tout un processus par lots.

Un *Job* est câblé avec une configuration :  
fichier de XML ou configuration Java.

Il combine plusieurs étapes (steps) ensemble  
qui appartiennent logiquement à un flux

Il permet la configuration des propriétés  
globales à toutes les étapes, comme les  
propriétés de redémarrage





# Configuration d'un job

---

Les principales propriétés d'un job sont donc :

- Son nom
- La définition et le séquençement des étapes
- Sa possibilité de redémarrage



# Configuration Java

---

Lors de la configuration Java, *Spring* met à disposition des builders pour l'instanciation d'un job.

Ex : *JobBuilderFactory*

@Bean

```
public Job footballJob() {  
    return this.jobBuilderFactory.get("footballJob")  
        .start(playerLoad())  
        .next(gameLoad())  
        .next(playerSummarization())  
        .end()  
        .build();  
}
```



# Configuration XML

---

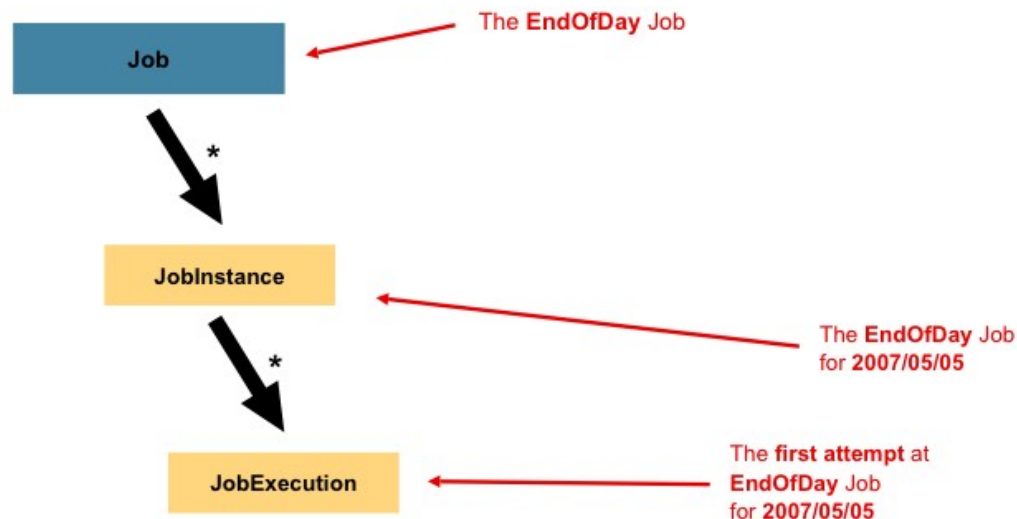
Avec la configuration XML, l'espace de nom *batch* permet de définir un job via la balise `<job>`

```
<job id="footballJob">  
  <step id="playerload" next="gameLoad"/>  
  <step id="gameLoad" next="playerSummarization"/>  
  <step id="playerSummarization"/>  
</job>
```

# JobInstance

**JobInstance** représente un démarrage du job

Une seule instance peut être démarrée à la fois mais il peut contenir plusieurs exécutions (*JobExecution*), si certaines échouent)





# *JobParameters*

---

***JobParameters*** encapsule un ensemble de paramètres utilisés pour démarrer un job.

Les paramètres peuvent être utilisés :

- Comme identification de l'instance
- Comme données de référence pendant l'exécution



# *JobExecution*

---

***JobExecution*** correspond à une tentative d'exécution d'un Job.

- Elle peut se terminer par un échec ou un succès

Le *JobInstance* correspondant est terminé si une exécution se termine avec succès.



# Données persistantes

---

*JobExecution* contient des propriétés persistantes indiquant ce qui s'est passé durant l'exécution :

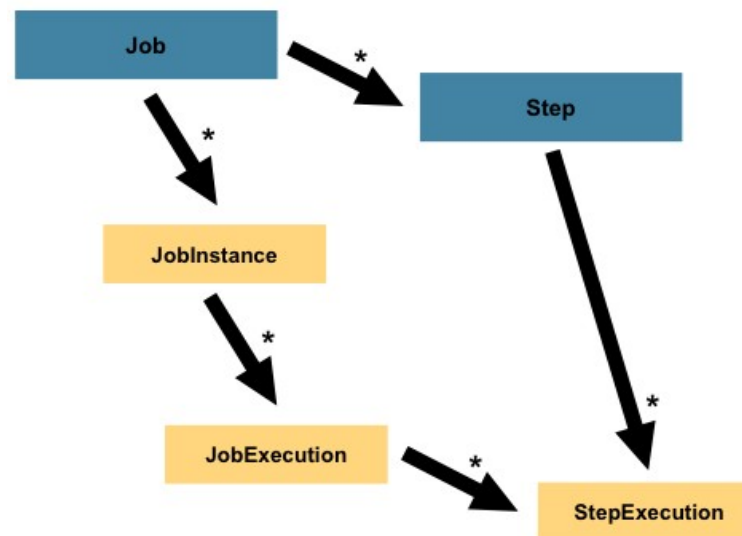
- ***Status*** (BatchStatus)
- ***createTime, startTime, endTime***
- ***exitStatus***
- ***lastUpdated***
- ***executionContext***
- ***failureExceptions***

# Step, StepExecution

**Step** encapsule une phase séquentielle indépendante d'un batch

Le contenu d'un step est à la discrétion du développeur, il peut être simple comme complexe

Un *Step* a un **StepExecution** en corrélation d'un *JobExecution*







# *StepExecution*

---

***StepExecution*** représente une tentative pour exécuter un *Step*.

Ses propriétés sont également persistantes :

- *Status, createTime, startTime, endTime, exitStatus, executionContext*
- ***readCount, writeCount***
- ***commitCount, rollbackCount***
- ***readSkipCount, processSkipCount, writeSkipCount, filterCount***

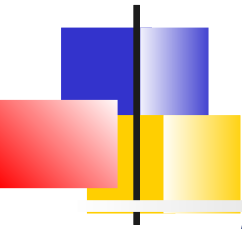


# *ExecutionContext*

---

***ExecutionContext*** représente une collection de paires clé / valeur persistantes

- Permet de stocker l'état persistant associé à un *StepExecution* ou un *JobExecution*
- Pour permettre un redémarrage, produire des statistiques, ...



# *JobRepository*

***JobRepository*** fournit des opérations CRUD pour les *JobExecution* et *StepExecution*

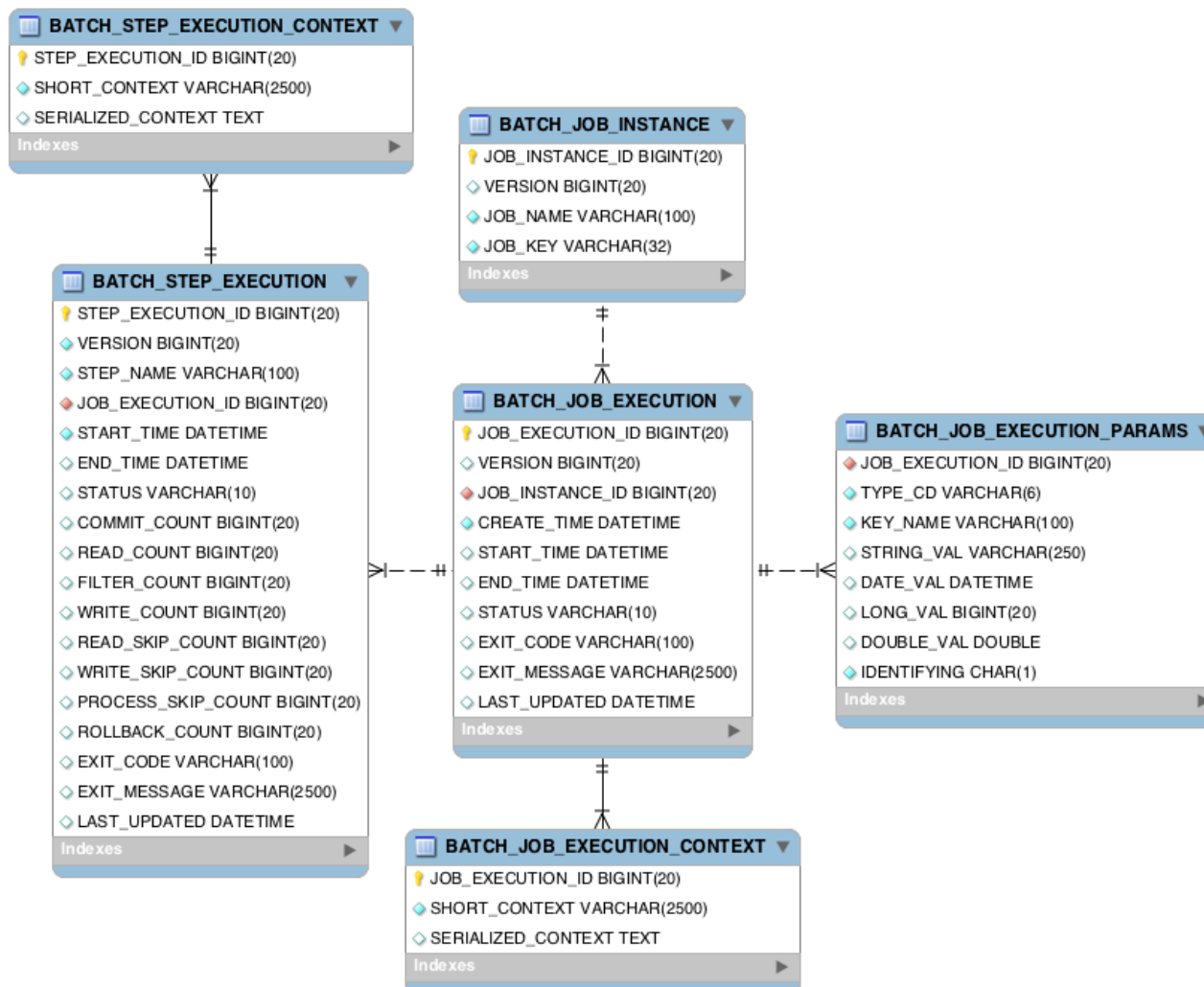
- Principalement utilisé par JobLauncher, Job et Step

Configuré en XML par

```
<job-repository id="jobRepository"/>
```

Et en Java, configuration automatique via  
*@EnableBatchProcessing*

# Schéma





# JobLauncher

---

***JobLauncher*** est une interface pour démarrer un Job avec un ensemble de *JobParameters*

```
public interface JobLauncher {  
  
    public JobExecution run(Job job, JobParameters jobParameters)  
        throws JobExecutionAlreadyRunningException,  
               JobRestartException,  
               JobInstanceAlreadyCompleteException,  
               JobParametersInvalidException;  
  
}
```



# Éléments des Steps

---

***ItemReader*** est une abstraction qui représente la récupération de l'entrée d'une étape, un élément à la fois.

***ItemWriter*** est une abstraction qui représente la sortie d'une étape, d'un lot ou d'un bloc d'éléments à la fois.

***ItemProcessor*** est une abstraction qui représente le traitement métier d'un élément.



# Batch Namespace

---

```
<beans:beans xmlns="http://www.springframework.org/schema/batch"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/batch
https://www.springframework.org/schema/batch/spring-batch.xsd">
<job id="ioSampleJob">
  <step id="step1">
    <tasklet>
      <chunk reader="itemReader" writer="itemWriter" commit-interval="2"/>
    </tasklet>
  </step>
</job>
</beans:beans>
```



# Spring Batch et Spring Boot

---

## **Rappels Spring Boot**

Dépendances pour SpringBatch  
Configuration par défaut et surcharge





# Introduction

---

Spring Boot a été conçu pour **simplifier le démarrage** et le développement de nouvelles applications Spring

- ne nécessite aucune configuration XML
- Dès la première ligne de code, on a une application fonctionnelle

=> Offrir une expérience de développement simplifiant à l'extrême l'utilisation des technologies existantes



# Essence

---

Spring Boot est un ensemble de bibliothèques qui sont exploitées par un système de build et de gestion de dépendances ( ***Maven*** ou ***Gradle*** )



# Auto-configuration

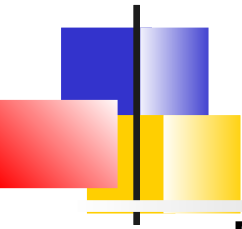
---

Le concept principal de *SpringBoot* est l'**auto-configuration**

*SpringBoot* est capable de détecter automatiquement la nature de l'application et de configurer les beans Spring nécessaires

- Cela permet de démarrer rapidement et de graduellement surcharger la configuration par défaut pour les besoins de l'application

Les mécanismes sont différents en fonction du langage : Groovy, Java ou Kotlin



# Java

## Gestion des dépendances

---

Dans un environnement Java, Spring Boot simplifie la gestion de dépendances et de leurs versions :

- Il organise les fonctionnalités de Spring en modules.  
=> Des groupes de dépendances peuvent être ajoutés à un projet en important des **"starter" modules**.
- Il fournit un POM parent dont les projets héritent qui gère les versions des dépendances.  
=> Le projet ne gère alors qu'un seul n° de version , celui de SpringBoot
- Il propose **"Spring Initializr"**, qui peut être utilisée via un navigateur ou un IDE, qui permet de générer des configurations Maven ou Gradle



# Starter Modules

---

***spring-boot-starter-web:*** librairies de Spring MVC + configuration automatique d'un serveur embarqué (Tomcat, Undertow, ...).

***spring-boot-starter-reactive-web:*** librairies Spring WebFlux + configuration automatique d'un serveur embarqué (Netty).

***spring-boot-starter-data-\* :*** Librairies d'accès aux données persistantes (JPA, NoSQL, SolR, ...). Facilite la configuration des sources de données et l'implémentation de la couche DAO

***spring-boot-starter-security*** : librairies de SpringSecurity + configuration simpliste de la sécurité

***spring-boot-starter-actuator*** : Permet l'exposition de points de surveillance via HTTP ou JMX (métriques de performances, sondes, audit sécurité, traces HTTP, ...).



# Structure projet

---

Aucune obligation mais des recommandations :

- Placer la classe *Main* dans le package racine
- L'annoter avec :
  - Les annotations
    - **@EnableAutoConfiguration**
    - **@ComponentScan**
    - **@Configuration**
  - Ou tout simplement :  
**@SpringBootApplication**



# Projet Java

---

```
package com.infoq.springboot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.web.bind.annotation.*;

@RestController
@SpringBootApplication
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello";
    }

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```



# *Jar* exécutable

---

La classe *Application* est exécutable, ce qui veut dire que l'application, et son conteneur embarqué, peuvent être démarrés en tant qu'application Java

- Les plugins Maven et Gradle de Boot permettent de produire un "fat jar" exécutable  
*mvn package,*  
*gradle build*





# Auto-configuration

---

**@EnableAutoConfiguration** permet de configurer automatiquement des beans Spring en fonction des dépendances qui ont été spécifiées.

- Possibilité de désactiver l'auto-configuration pour certaines parties de l'application.

Ex :

```
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
```



# @Configuration

---

**@Configuration** indique à Spring que le classe peut définir des beans Spring

- La classe *Main* peut être un bon emplacement pour la configuration
- Mais celle-ci peut être dispersée dans plusieurs autres classes
- L'annotation **@Import** peut être utilisée pour importer les autres classes de configuration



# Beans et Injection de dépendance

---

Il est possible d'utiliser les différentes techniques de Spring pour définir les beans et leurs injections de dépendances.

La technique la plus simple est généralement la combinaison de :

- L'annotation **@ComponentScan** afin que *Spring* trouve les beans (Inclut dans *@SpringBootApplication*)
- L'annotation **@Autowired** dans le constructeur d'un bean ou sur une déclaration
- L'utilisation de l'injection implicite, attribut final + paramètre du constructeur
- Les annotations **@Component**, **@Service**, **@Repository**, **@Controller** qui permettent de définir des beans



# Exemple

---

```
package com.example.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class DatabaseAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

    @Autowired
    public DatabaseAccountService(RiskAssessor riskAssessor) {
        this.riskAssessor = riskAssessor;
    }

    // ...

}
```



# Personnalisation de la configuration par défaut

---

La configuration par défaut peut être surchargée par différents moyens

- Des **fichiers de configuration externe** (*.properties* ou *.yml*) permettent de fixer des valeurs des variables de configuration.  
On peut mettre en place différents fichiers en fonction de profils (correspondant aux environnements)
- Des classes utilitaires Spring **\*Configurer** ou **\*Customizer** permettant de surcharger la configuration par défaut via l'API
- Utiliser des **classes spécifiques** au bean que l'on veut surcharger (exemple *AuthenticationManagerBuilder*)
- La **définition de Beans** remplaçant les beans par défaut
- La **désactivation** de l'auto-configuration



# Spring Batch et Spring Boot

---

Rappels Spring Boot  
**Spring Boot et SpringBatch**



# starter

---

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

## Inclut :

- Starter jdbc
- Batch Core
- Starter-test
- batch-test



# Auto-configuration

---

Par défaut, un Runner est créé et toutes les jobs présents dans le contexte seront exécutées au démarrage.

- Désactiver avec  
`spring.batch.job.enabled = false`
- Les noms de Job à exécuter peuvent être fournis via:  
`spring.batch.job.names = job1, job2`  
Dans ce cas, le Runner trouvera d'abord les jobs enregistrés en tant que Beans, puis ceux du *JobRegistry* existant.





# Beans

---

`org.springframework.batch.core.scope.StepScope@6ca372ef`

`org.springframework.batch.core.scope.JobScope@3ebe4ccc`

`org.springframework.batch.core.configuration.annotation.JobBuilderFactory@135a8c6f`

`org.springframework.batch.core.configuration.annotation.StepBuilderFactory@6419a0e1`

`org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration$DataSourceInitializerConfiguration@280d4882`

`org.springframework.boot.autoconfigure.batch.BatchDataSourceInitializer@44af588b`

`org.springframework.boot.autoconfigure.batch.BatchConfigurerConfiguration$JdbcBatchConfiguration@3d19d85`

`org.springframework.boot.autoconfigure.batch.BasicBatchConfigurer@2ae62bb6`

`org.springframework.boot.autoconfigure.batch.BatchConfigurerConfiguration@68ed3f30`

`org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration@56b859a6`

`org.springframework.boot.autoconfigure.batch.JobLauncherApplicationRunner@2f4bala`

`org.springframework.boot.autoconfigure.batch.JobExecutionExitCodeGenerator@4397a639`

`org.springframework.batch.core.launch.support.SimpleJobOperator@7dffda8b`

`org.springframework.boot.autoconfigure.batch.BatchProperties@3d1f558a`



# Configuration de jobs

---

**Jobs**

Steps

Configuration par défaut et surcharge



# Introduction

---

Considérations à prendre en compte, lors de la configuration :

- Comment le job sera lancé ?
- Quelles méta-données seront stockées durant l'exécution ?



# Configuration basique

---

Java

@Bean

```
public Job footballJob() {  
    return this.jobBuilderFactory.get("footballJob")  
        .start(playerLoad())  
        .next(gameLoad())  
        .next(playerSummarization())  
        .end().build();  
}
```

XML

```
<job id="footballJob">  
    <step id="playerload" parent="s1" next="gameLoad"/>  
    <step id="gameLoad" parent="s2" next="playerSummarization"/>  
    <step id="playerSummarization" parent="s3"/>  
</job>
```

Les 2 configurations supposent un bean de type  
*JobRepository* (id=*jobRepository*)



# Redémarrage

---

Le lancement d'un Job est considéré comme un 'redémarrage' si un *JobExecution* existe déjà ce Job.

Il est possible d'autoriser ou d'interdire les redémarrage.

```
this.jobBuilderFactory.get("footballJob")  
.preventRestart()
```

```
<job id="footballJob" restartable="false">
```



# *JobRestartException*

---

```
Job job = new SimpleJob();
job.setRestartable(false);
JobParameters jobParameters = new JobParameters();
JobExecution firstExecution =
    jobRepository.createJobExecution(job, jobParameters);
jobRepository.saveOrUpdate(firstExecution);

try {
    jobRepository.createJobExecution(job, jobParameters);
    fail();
} catch (JobRestartException e) {
    // expected
}
```



# JobListener

---

Des *JobListeners* peuvent être ajoutés à un Job

```
public interface JobExecutionListener {  
    void beforeJob(JobExecution jobExecution);  
    // Appelée quelque soit l'issue du job  
    void afterJob(JobExecution jobExecution);  
}
```

----

Configuration :

```
this.jobBuilderFactory.get("footballJob")  
    .listener(sampleListener())
```

```
<job id="footballJob">
```

---

```
    <listeners>  
        <listener ref="sampleListener"/>  
    </listeners>  
</job>
```



# Héritage d'un job avec XML

---

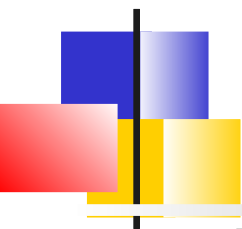
Même avec une configuration XML, on peut hériter d'une configuration parente<sup>1</sup>.

```
<job id="baseJob" abstract="true">
  <listeners>
    <listener ref="listenerOne"/>
  </listeners>
</job>
```

```
<job id="job1" parent="baseJob">
  <step id="step1" parent="standaloneStep"/>
  <listeners merge="true">
    <listener ref="listenerTwo"/>
  </listeners>
</job>
```

*1. En java un job peut tout simplement « extends » un Job parent*






# Validateur de paramètres

---

Un job peut déclarer un bean responsable de valider les paramètres

Un *DefaultJobParametersValidator* est disponible, il peut combiner les contraintes de paramètres obligatoires et facultatifs simples.

Pour des contraintes plus complexes, on fournit sa propre implémentation de *JobParametersValidator*



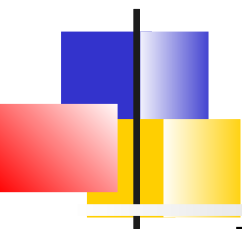
# Java et Configuration par défaut

L'annotation **@EnableBatchProcessing** fournit une configuration de base pour la Jobs.

Dans cette configuration de base, une instance de *StepScope* est créée en plus d'un certain nombre de beans :

- **JobRepository (jobRepository)** : Support persistant
- **JobLauncher (jobLauncher)** : Contrôleur de jobs
- **JobRegistry (jobRegistry)** : Service de noms des jobs
- **PlatformTransactionManager (transactionManager)** : Gestionnaire de transaction
- **JobBuilderFactory (jobBuilders)** : Permet de configurer et instancier un job
- **StepBuilderFactory (stepBuilders)** : Permet de configurer et instancier une step

L'implémentation par défaut du *JobRepository* nécessite la définition d'un bean de type *DataSource*



# Configuration JobRepository

---

La configuration du *jobRepository* est nécessaire.

- Soit on profite de la configuration défaut Java, soit on configure explicitement en XML

Les options de configuration sont :

- La source de données
- Le gestionnaire de transaction
- Le niveau d'isolation pour la création de job (Par défaut `SERIALIZABLE`)
- Le préfixe des tables (Par défaut `BATCH`)
- La longueur maximale des colonnes *VARCHAR*



# Configurations des options

---

## XML

```
<job-repository id="jobRepository"
data-source="dataSource"
transaction-manager="transactionManager"
isolation-level-for-create="SERIALIZABLE"
table-prefix="BATCH_"
Max-varchar-length="1000"/>
```

## Java

### // Surcharge de BatchConfigurer

```
@Override
protected JobRepository createJobRepository() throws Exception {
    JobRepositoryFactoryBean factory = new JobRepositoryFactoryBean();
    factory.setDataSource(dataSource);
    factory.setTransactionManager(transactionManager);
    factory.setIsolationLevelForCreate("ISOLATION_SERIALIZABLE");
    factory.setTablePrefix("BATCH_");
    factory.setMaxVarCharLength(1000);
    return factory.getObject();
}
```



# Configuration des transactions

---

Si l'espace de noms ou le *FactoryBean* fourni est utilisé, l'aspect transactionnel est automatiquement ajouté aux méthodes du repository.

Le niveau d'isolation lors de la création est configuré séparément. Il permet de s'assurer si on tente de démarrer en même temps 2 fois le même jobs. Un seul sera effectivement démarré.

- Le niveau par défaut *SERIALIZABLE* offre une garantie complète



# Repository mémoire

---

Spring batch fournit une version *Map* (mémoire) de *JobRepository* pour le prototypage et les tests.

```
// BatchConfigurer
@Override
protected JobRepository createJobRepository() throws
    Exception {
    MapJobRepositoryFactoryBean factory = new
        MapJobRepositoryFactoryBean();
    factory.setTransactionManager(transactionManager);
    return factory.getObject();
}
```



# Type de BD

---

Avec *JobRepositoryFactoryBean*, il est possible de préciser le type de la base de données, par exemple

```
factory.setDatabaseType("db2");
```

Sinon, il essaie de détecter automatiquement le type de base de données à partir de *DataSource*

Les principales différences entre les plates-formes concernent l'incrémentation automatique des clés primaires

Si la base n'est pas supportée, implémenter soi-même *incrementerFactory*



# *JobLauncher*

---

Avec *@EnableBatchProcessing*, un ***jobLauncher*** est fourni automatiquement. (SimpleJobLauncher)

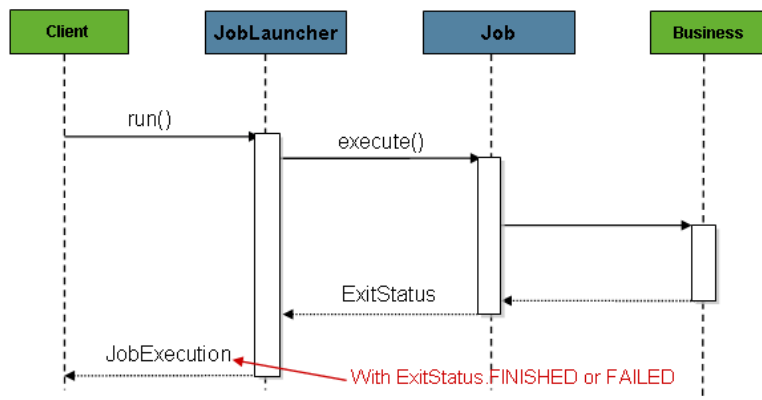
– Sa seule dépendance est le JobRepository.

```
// BatchConfigurer
@Override
protected JobLauncher createJobLauncher() throws Exception {
    SimpleJobLauncher jobLauncher = new SimpleJobLauncher();
    jobLauncher.setJobRepository(jobRepository);
    jobLauncher.afterPropertiesSet();
    return jobLauncher;
}
```

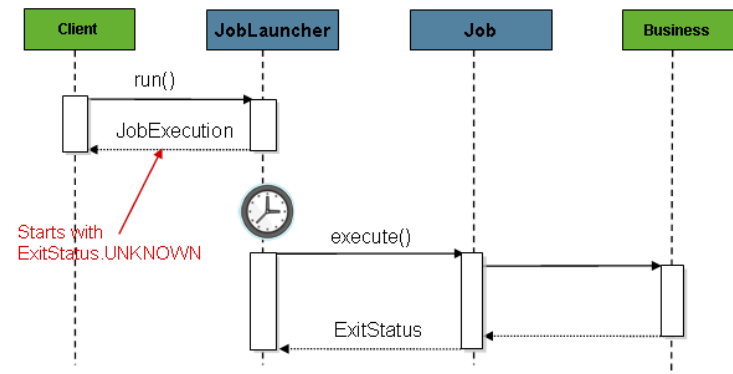


# Synchrone / Asynchrone

Par défaut synchrone, mais peut se configurer dans un modèle asynchrone



28



30



# Configuration asynchrone

---

Un *JobLauncher* peut être configuré pour de l'asynchrone grâce à un ***TaskExecutor***

L'interface définit une méthode :  
`void execute(Runnable task)`

Ex :

```
jobLauncher.setTaskExecutor(  
    new SimpleAsyncTaskExecutor()  
);
```



# Execution d'un job

---

Le démarrage du Job peut se faire de différentes façons.  
Les cas typiques sont :

- Via un scheduler
- Via une application Web
- ..

Cela consiste généralement à :

- Charger le bon ApplicationContext
- Instancier les JobParameters, en parsant la ligne de commande ou les paramètres HTTP
- Localiser le job en fonction des arguments
- Utiliser le JobLauncher fourni par le contexte pour démarrer le job.



# *CommandLineJobRunner*

---

Spring Batch fournit une implémentation permettant de démarrer un job via une ligne de commande :

## ***CommandLineJobRunner***

Il prend en argument :

- Un fichier XML ou une classe de configuration Java permettant de charger *l'ApplicationContext*
- Le nom du job
- Les paramètres du Job

Exemples :

```
$> java CommandLineJobRunner endOfDayJob.xml endOfDay \  
schedule.date(date)=2007/05/05  
$> java CommandLineJobRunner io.spring.EndOfDayJobConfiguration  
endOfDay \  
schedule.date(date)=2007/05/05
```



# Contexte SpringBoot

---

```
@SpringBootApplication
@EnableBatchProcessing
public class SpringBootBatchProcessingApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBootBatchProcessingApplication.class, args);
    }
}
...
mvn clean package
...
java -jar my
```



# *ExitCode*

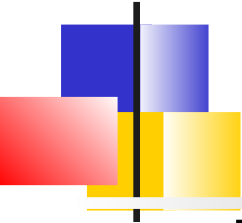
---

Le traitement batch géré par des schedulers nécessite de retourner des codes numériques :

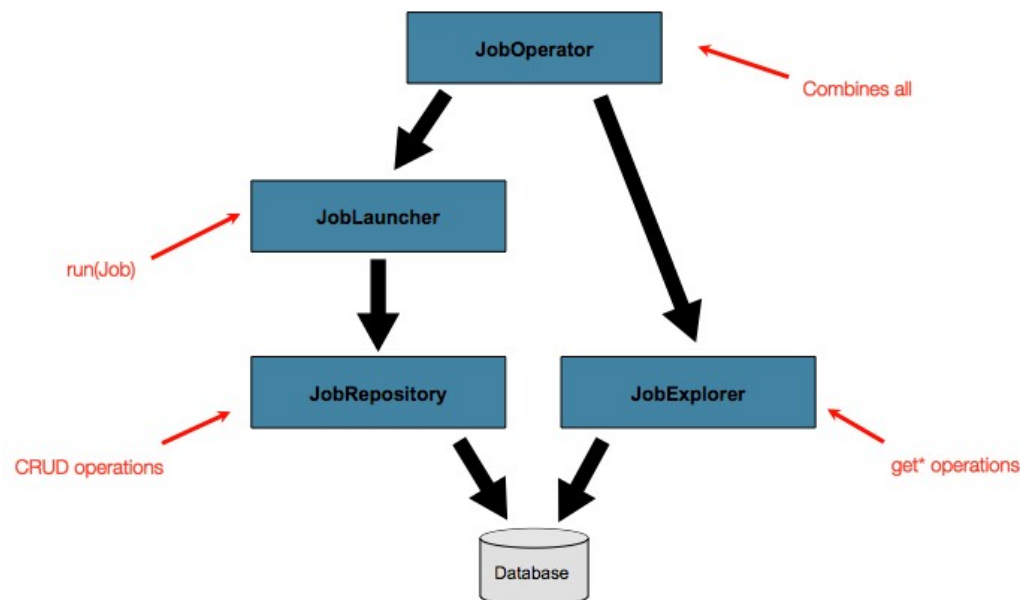
- En général 0 = OK et 1 = Error
- Mais on peut avoir plus de valeurs de retour possible

Spring Batch permet d'encapsuler le code de sortie dans un objet ***ExitStatus***

- Sa propriété de type String est converti par un bean ***ExitCodeMapper***
- L'implémentation par défaut *SimpleJvmExitCodeMapper* retourne ;
  - 0 pour succès
  - 1 pour les erreurs génériques
  - 2 pour les erreurs de job runner comme par exemple (Impossible de trouver le job)
- Il est possible d'implémenter son propre *ExitCodeMapper*



Lorsque l'on doit gérer de nombreux jobs et de contraintes de scheduling plus complexe, les interfaces **JobOperator** et **JobExplorer** permettent de contrôler les méta-données associées





# Interface *JobExplorer*

---

***JobExplorer*** permet de requêter vers le JobRepository les exécutions existantes

```
public interface JobExplorer {  
    List<JobInstance> getJobInstances(String jobName, int start, int count);  
    JobExecution getJobExecution(Long executionId);  
    StepExecution getStepExecution(Long jobExecutionId, Long stepExecutionId);  
    JobInstance getJobInstance(Long instanceId);  
    List<JobExecution> getJobExecutions(JobInstance jobInstance);  
    Set<JobExecution> findRunningJobExecutions(String jobName);  
}
```





# Configuration

---

## XML

```
<bean id="jobExplorer"  
    class="org.spr...JobExplorerFactoryBean"  
    p:dataSource-ref="dataSource" />
```

## Java

```
// BatchConfigurer  
@Override  
public JobExplorer getJobExplorer() throws Exception {  
    JobExplorerFactoryBean factoryBean = new  
        JobExplorerFactoryBean();  
    factoryBean.setDataSource(this.dataSource);  
    return factoryBean.getObject();  
}
```



# *JobRegistry*

---

Un ***JobRegistry*** n'est pas obligatoire, mais peut être utile pour suivre les jobs disponibles dans le contexte.

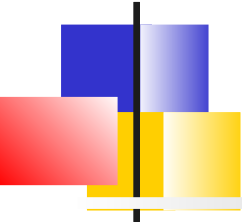
Une seule implémentation, basée sur une Map (nom du job, instance) est fournie par le framework



# *JobOperator*

---

L'interface ***JobOperator*** permet d'effectuer des tâches de surveillance courantes telles que l'arrêt, le redémarrage ou visualiser le résumé d'un job



# Interface *JobOperator*

---

```
public interface JobOperator {  
    List<Long> getExecutions(long instanceId)  
    List<Long> getJobInstances(String jobName, int start, int count)  
    Set<Long> getRunningExecutions(String jobName)  
    String getParameters(long executionId)  
    Long start(String jobName, String parameters)  
    Long restart(long executionId)  
    Long startNextInstance(String jobName)  
    boolean stop(long executionId)  
    String getSummary(long executionId)  
    Map<Long, String> getStepExecutionSummaries(long executionId)  
    Set<String> getJobNames();  
}
```



# Exemple : Arrêt d'un Job

---

```
// dès que le contrôle est retourné au framework,  
// Le statut du StepExecution devient  
    BatchStatus.STOPPED,  
// Puis celui de JobExecution Set<Long> executions  
    = jobOperator.getRunningExecutions("sampleJob");  
jobOperator.stop(executions.iterator().next());
```



# Configuration XML

---

```
<bean id="jobOperator" class="org.spr...SimpleJobOperator">
  <property name="jobExplorer">
    <bean class="org.spr...JobExplorerFactoryBean">
      <property name="dataSource" ref="dataSource" />
    </bean>
  </property>
  <property name="jobRepository" ref="jobRepository" />
  <property name="jobRegistry" ref="jobRegistry" />
  <property name="jobLauncher" ref="jobLauncher" />
</bean>
```



# Configuration Java

---

@Bean

```
public SimpleJobOperator jobOperator(JobExplorer jobExplorer,  
JobRepository jobRepository, JobRegistry jobRegistry) {  
    SimpleJobOperator jobOperator = new SimpleJobOperator();  
    jobOperator.setJobExplorer(jobExplorer);  
    jobOperator.setJobRepository(jobRepository);  
    jobOperator.setJobRegistry(jobRegistry);  
    jobOperator.setJobLauncher(jobLauncher);  
    return jobOperator;  
}
```



# JobParametersIncrementer

---

La méthode *startNextInstance* utilise le ***JobParametersIncrementer*** associé au Job pour forcer une nouvelle instance

L'implémentation est responsable de fournir les paramètres de la prochaine instance de job

```
public interface JobParametersIncrementer {  
    JobParameters getNext(JobParameters parameters);  
}
```





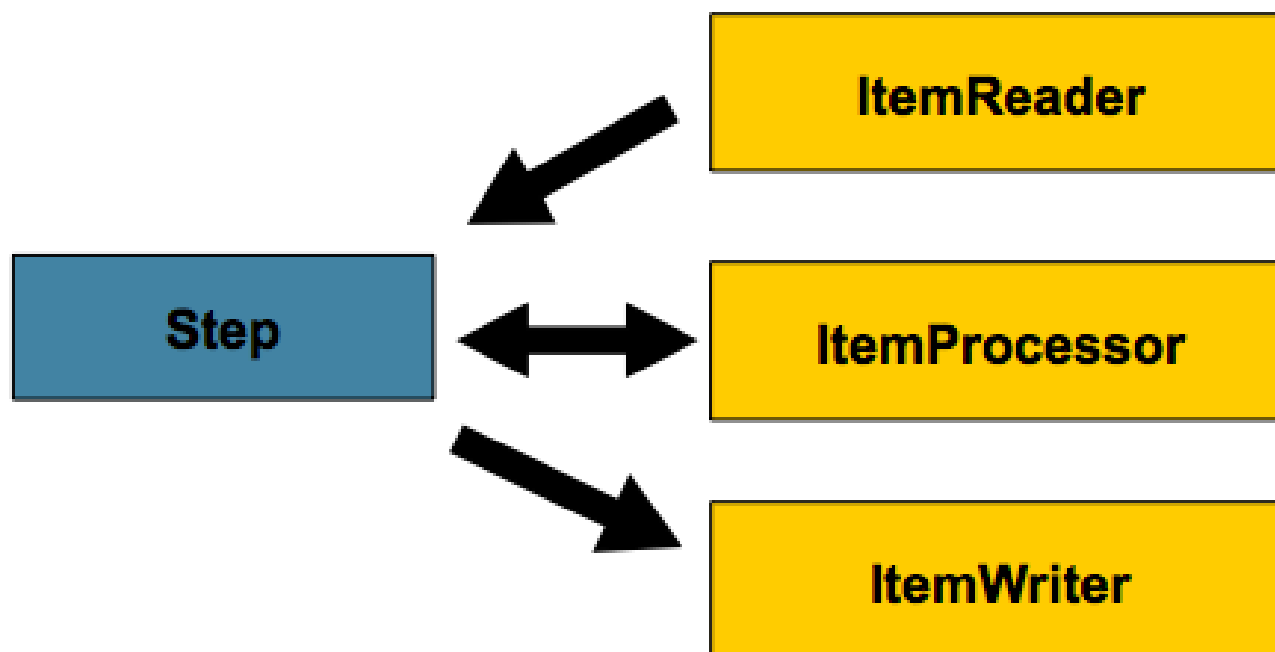
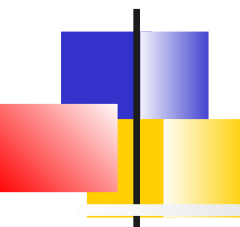
# Configuration des steps

---

Jobs

**Steps**

Configuration par défaut et surcharge





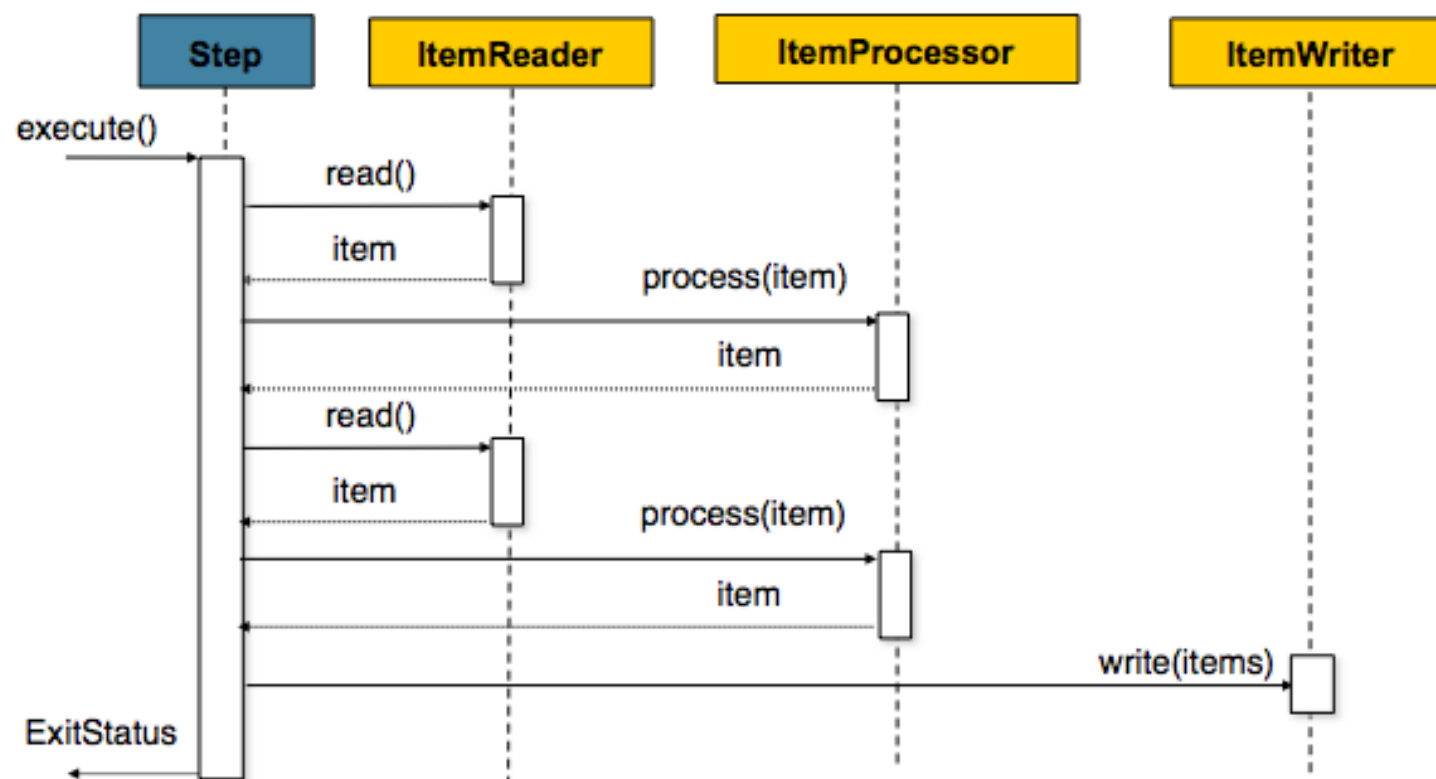
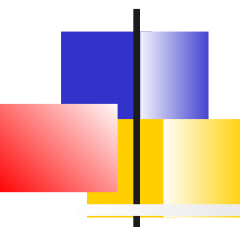
# Chunk-oriented

---

Le traitement orienté chunk (morceau) fait référence à la lecture des données une par une et à la création de «morceaux» qui sont écrits en une transaction.

=> Un élément est lu à partir d'un ItemReader, remis à un ItemProcessor puis agrégé.

=> Une fois que le nombre d'éléments lus est égal à l'intervalle de validation, le bloc entier est écrit par ItemWriter, puis la transaction est validée





```
List items = new ArrayList();
for(int i = 0; i < commitInterval; i++){
    Object item = itemReader.read()
    Object processedItem =
        itemProcessor.process(item);
    items.add(processedItem);
}
itemWriter.write(items);
```



# Configuration XML

---

```
<job id="sampleJob" job-repository="jobRepository">
  <step id="step1">
    <tasklet transaction-manager="transactionManager">
      <chunk reader="itemReader" writer="itemWriter"
        commit-interval="10"/>
    </tasklet>
  </step>
</job>
```



# Configuration Java

---

@Bean

```
public Job sampleJob(JobRepository jobRepository, Step sampleStep) {  
    return this.jobBuilderFactory.get("sampleJob")  
        .repository(jobRepository)  
        .start(sampleStep)  
        .build();  
}
```

@Bean

```
public Step sampleStep(PlatformTransactionManager transactionManager) {  
    return this.stepBuilderFactory.get("sampleStep")  
        .transactionManager(transactionManager)  
        .<String, String>chunk(10)  
        .reader(itemReader())  
        .writer(itemWriter())  
        .build();  
}
```



# Dépendances requises pour une Step

---

***reader***: ItemReader qui fournit des éléments à traiter.

***writer***: ItemWriter qui traite les éléments fournis par ItemReader.

***transaction-manager*** : Gestionnaire de transaction qui commence et valide les transactions.

***job-repository*** : Le *JobRepository* qui stocke périodiquement *StepExecution* et *ExecutionContext* pendant le traitement (juste avant la validation).

***commit-interval* / *chunk***: le nombre d'éléments à traiter avant que la transaction ne soit committée.





# Héritage

---

Si un groupe de steps partage des configurations similaires, il est utile de définir une étape «parent» à partir de laquelle les étapes concrètes peuvent hériter

```
<step id="parentStep">
  <tasklet allow-start-if-complete="true">
    <chunk reader="itemReader" writer="itemWriter" commit-interval="10"/>
  </tasklet>
</step>
```

```
<step id="concreteStep1" parent="parentStep">
  <tasklet start-limit="5">
    <chunk processor="itemProcessor" commit-interval="5"/>
  </tasklet>
</step>
```



# Abstract Step

---

```
<!-- Configuration incomplète (pas de reader, pas de writer)  
Alors la step est abstraite -->
```

```
<step id="abstractParentStep" abstract="true">  
  <tasklet>  
    <chunk commit-interval="10"/>  
  </tasklet>  
</step>
```

```
<step id="concreteStep2" parent="abstractParentStep">  
  <tasklet>  
    <chunk reader="itemReader" writer="itemWriter"/>  
  </tasklet>  
</step>
```



# Fusion de liste

---

```
<step id="listenersParentStep" abstract="true">
  <listeners>
    <listener ref="listenerOne"/>
  </listeners>
</step>
```

```
<!-- concreteStep3 a 2 listeners -->
```

```
<step id="concreteStep3" parent="listenersParentStep">
  <tasklet>
    <chunk reader="itemReader" writer="itemWriter" commit-interval="5"/>
  </tasklet>
  <listeners merge="true">
    <listener ref="listenerTwo"/>
  </listeners>
</step>
```



# Redémarrage des steps

---

Il est possible de positionner des contraintes sur les steps lors d'un redémarrage de job

- Limiter le nombre d'exécution d'une step :  
`<tasklet start-limit="1">`  
`.startLimit(1)`
- Redémarrer systématiquement une step terminée quelque soit son statut  
`<tasklet allow-start-if-complete="true">`  
`.allowStartIfComplete(true)`



# Example

---

```
<job id="footballJob" restartable="true">
  <step id="playerload" next="gameLoad">
    <tasklet>
      <chunk reader="playerFileItemReader" writer="playerWriter"
        commit-interval="10" />
    </tasklet>
  </step>
  <step id="gameLoad" next="playerSummarization">
    <tasklet allow-start-if-complete="true">
      <chunk reader="gameFileItemReader" writer="gameWriter"
        commit-interval="10"/>
    </tasklet>
  </step>
  <step id="playerSummarization">
    <tasklet start-limit="2">
      <chunk reader="playerSummarizationSource" writer="summaryWriter"
        commit-interval="10"/>
    </tasklet>
  </step>
</job>
```



# Explication

---

Dans l'exemple précédent :

- l'étape *playerLoad* peut être démarré un nombre illimité de fois et, s'il s'est terminé normalement, il est ignoré
- l'étape *gameLoad* est redémarré à chaque fois
- l'étape *playerSummarization* est redémarré au maximum 2 fois



# skip

---

Il est possible d'ignorer un step en cas d'erreur

If faut configurer les exceptions et leur nombre max qui ne font pas échouer l'étape mais qui saute juste l'item en cours de traitement

```
@Bean
public Step step1() {
    return this.stepBuilderFactory.get("step1")
        .<String, String>chunk(10)
        .reader(flatFileItemReader())
        .writer(itemWriter())
        .faultTolerant()
        .skipLimit(10)
        .skip(FlatFileParseException.class)
        .build();
}
```



# Retry

---

Il est possible de retenter de traiter un item lors d'une exception particulière

```
@Bean
public Step step1() {
    return this.stepBuilderFactory.get("step1")
        .<String, String>chunk(2)
        .reader(itemReader())
        .writer(itemWriter())
        .faultTolerant()
        .retryLimit(3)
        .retry(DeadlockLoserDataAccessException.class)
        .build();
}
```





# Contrôle du rollback

---

Par défaut, indépendamment de la configuration de *retry* et *skip*, toutes les exceptions lancées à partir de *ItemWriter* provoquent un rollback de la transaction.

Il est possible de définir les exceptions qui ne provoquent pas de rollback

```
@Bean
public Step step1() {
    return this.stepBuilderFactory.get("step1")
        .<String, String>chunk(2)
        .reader(itemReader())
        .writer(itemWriter())
        .faultTolerant()
        .noRollback(ValidationException.class)
        .build();
}
```



# Attributs de transaction

---

Les attributs de transaction peuvent être utilisés pour contrôler les paramètres d'isolation, de propagation et de timeout.

```
<tasklet>  
  <chunk reader="itemReader" writer="itemWriter" commit-  
    interval="2"/>  
  <transaction-attributes isolation="DEFAULT"  
    propagation="REQUIRED" timeout="30"/>  
</tasklet>
```



# Attributs de transaction (Java)

---

```
@Bean
public Step step1() {
    DefaultTransactionAttribute attribute = new DefaultTransactionAttribute();
    attribute.setPropagationBehavior(Propagation.REQUIRED.value());
    attribute.setIsolationLevel(Isolation.DEFAULT.value());
    attribute.setTimeout(30);

    return this.stepBuilderFactory.get("step1")
        .<String, String>chunk(2)
        .reader(itemReader())
        .writer(itemWriter())
        .transactionAttribute(attribute)
        .build();
}
```



# Listeners

---

Il est possible d'associer des listeners des évènements liés aux steps

Les classes qui implémentent une extension de l'interface ***StepListener*** peuvent être configurées comme listeners :

- Via l'élément *<listeners>*
- Via la méthode *listener()*



# StepExecutionListener

---

***StepExecutionListener*** permet une notification avant le démarrage d'une étape et après sa fin, qu'elle se soit terminée normalement ou qu'elle ait échoué.

```
public interface StepExecutionListener extends StepListener {  
    void beforeStep(StepExecution stepExecution);  
    ExitStatus afterStep(StepExecution stepExecution);  
}
```

Annotations : @BeforeStep, @AfterStep



# *ChunkListener*

---

Un ***ChunkListener*** peut être utilisé pour exécuter une logique avant ou après le traitement d'un chunk

```
public interface ChunkListener extends StepListener {  
    void beforeChunk(ChunkContext context);  
    void afterChunk(ChunkContext context);  
    void afterChunkError(ChunkContext context);  
}
```

Annotations : *@BeforeChunk*,  
*@AfterChunk*, *@AfterChunkError*



# *ItemReadListener*

---

***ItemReaderListener*** est à l'écoute des opérations de lecture d'item. Il est assez adapté pour traiter les erreurs de lecture.

```
public interface ItemReadListener<T> extends StepListener {  
    void beforeRead();  
    void afterRead(T item);  
    void onReadError(Exception ex);  
}
```

Annotations : *@BeforeRead*, *@AfterRead*,  
*@OnReadError*



# *ItemProcessListener*

---

***ItemProcessListener*** écoute le traitement d'un item

```
public interface ItemProcessListener<T, S> extends  
    StepListener {  
    void beforeProcess(T item);  
    void afterProcess(T item, S result);  
    void onProcessError(T item, Exception e);  
}
```

Annotations : *@BeforeProcess*,  
*@AfterProcess*, *@OnProcessError*





# *ItemWriteListener*

---

***ItemWriteListener*** écoute l'écriture d'un lot d'items

```
public interface ItemWriteListener<S> extends StepListener {  
    void beforeWrite(List<? extends S> items);  
    void afterWrite(List<? extends S> items);  
    void onWriteError(Exception exception, List<? extends S>  
        items);  
}
```

Annotations : *@BeforeWrite*,  
*@AfterWrite*, *@OnWriteError*



# *SkipListener*

---

***SkipListener*** permet d'être au courant lorsque des items sont ignorés

```
public interface SkipListener<T,S> extends StepListener {  
    void onSkipInRead(Throwable t);  
    void onSkipInProcess(T item, Throwable t);  
    void onSkipInWrite(S item, Throwable t);  
}
```

Annotations : *@OnSkipInRead*,  
*@OnSkipInWrite*, *@OnSkipInProcess*



# TaskletStep

---

Il est possible d'exécuter des étapes qui ne contiennent pas *d'ItemReader* ou *d'ItemWriter*

**Tasklet** est une interface simple qui a une méthode, `execute`,

Elle est appelée à plusieurs reprises par le **TaskletStep** jusqu'à ce qu'il renvoie *RepeatStatus.FINISHED* ou lève une exception pour signaler un échec.

Chaque appel à un *Tasklet* est encapsulé dans une transaction.



# Configuration Tasklet

---

## XML

```
<step id="step1">  
  <tasklet ref="myTasklet"/>  
</step>
```

## Java

@Bean

```
public Step step1() {  
    return this.stepBuilderFactory.get("step1")  
        .tasklet(myTasklet())  
        .build();  
}
```



# *TaskletAdpater*

---

Une implémentation de *Tasklet* :  
***TaskletAdpater*** permet d'utiliser une  
classe existante

```
@Bean
public MethodInvokingTaskletAdapter myTasklet() {
    MethodInvokingTaskletAdapter adapter =
        new MethodInvokingTaskletAdapter();

    adapter.setTargetObject(fooDao());
    adapter.setTargetMethod("updateFoo");
    return adapter;
}
```



# Contrôle du *flow*

---

*SpringBatch* donne la possibilité de contrôler l'enchaînement des étapes

Comme par exemple :

- Indiquer que l'échec d'une étape ne fait pas échouer le job
- En fonction de l'issue d'une exécution, déterminer quelle étape s'exécute
- En fonction de la configuration d'un groupe d'étapes, certaines étapes ne sont pas exécutées.
- ...



# Flow séquentiel

---

L'enchaînement le plus simple et d'exécuter séquentiellement toutes les étapes du job

```
@Bean
public Job job() {
    return this.jobBuilderFactory.get("job")
        .start(stepA())
        .next(stepB())
        .next(stepC())
        .build();
}
```



# Flow conditionnel

---

En fonction de ***l'ExitStatus*** d'une step, on peut déclencher

- Une step particulière
- Un arrêt du job

La configuration s'effectue avec la syntaxe on qui prend une valeur d'un ExitStatus ou une expression avec les caractères wildcard \* ou ?





# Configuration XML

---

```
<job id="job">
  <step id="stepA" parent="s1">
    <next on="*" to="stepB" />
    <next on="FAILED" to="stepC" />
  </step>
  <step id="stepB" parent="s2" next="stepC" />
  <step id="stepC" parent="s3" />
</job>
```



# Configuration Java

---

@Bean

```
public Job job() {  
    return this.jobBuilderFactory.get("job")  
        .start(stepA())  
        .on("*").to(stepB())  
        .from(stepA()).on("FAILED").to(stepC())  
        .end()  
        .build();  
}
```



# Compléments

---

- Si les résultats de l'exécution de l'étape n'est pas couvert par la configuration , alors le framework lève une exception et le job échoue
- Le framework ordonne automatiquement les transitions de la plus spécifique à la moins spécifique.
- Par défaut *ExitStatus* est égal à *BatchStatus* (énumération), mais il est possible de définir ses propres *ExitStatus* et de les configurer dans les transitions

```
public class SkipCheckingListener extends StepExecutionListenerSupport {  
    public ExitStatus afterStep(StepExecution stepExecution) {  
        ...  
        return new ExitStatus("COMPLETED WITH SKIPS");  
    }  
}
```



# Fin à une étape

---

Il est possible de définir une transition de fin.  
Dans ce cas, le batch s'arrête et a le statut **COMPLETED** (Il ne peut pas être redémarré)

```
@Bean
public Job job() {
    return this.jobBuilderFactory.get("job")
        .start(step1())
        .next(step2())
        .on("FAILED").end()
        .from(step2()).on("*").to(step3())
        .end()
        .build();
}
```



# Echouer le job à une étape

---

On peut configurer une transition afin qu'elle fasse échouer le job. Dans ce cas le Job a un *BatchStatus* de **FAILED** et peut être démarré.

```
<step id="step1" parent="s1" next="step2">
<step id="step2" parent="s2">
  <!-- ExitStatus=EARLY_TERMINATION, BatchStatus FAILED -->
  <fail on="FAILED" exit-code="EARLY TERMINATION"/>
  <next on="*" to="step3"/>
</step>
<step id="step3" parent="s3">
```



# Arrêter un job

---

Configurer un job afin qu'il s'arrête à une étape particulière lui donne un BatchStatus de **STOPPED**. Il peut être continué à une étape particulière

```
@Bean
public Job job() {
    return this.jobBuilderFactory.get("job")
        .start(step1()).on("COMPLETED").stopAndRestart(step2())
        .end()
        .build();
}
```



# *JobExecutionDecider*

---

Il est possible également de fournir un ***JobExecutionDecider*** pour implémenter des conditions plus complexe de séquencement

```
public class MyDecider implements JobExecutionDecider {  
    public FlowExecutionStatus decide(JobExecution jobExecution,  
                                     StepExecution stepExecution) {  
        String status;  
        if (someCondition()) {  
            status = "FAILED";  
        }  
        else {  
            status = "COMPLETED";  
        }  
        return new FlowExecutionStatus(status);  
    }  
}
```



# Configuration

---

```
<job id="job">
  <step id="step1" parent="s1" next="decision" />
  <decision id="decision" decider="decider">
    <next on="FAILED" to="step2" />
    <next on="COMPLETED" to="step3" />
  </decision>
  <step id="step2" parent="s2" next="step3"/>
  <step id="step3" parent="s3" />
</job>
<beans:bean id="decider" class="com.MyDecider"/>
---
@Bean
public Job job() {
  return this.jobBuilderFactory.get("job")
    .start(step1())
    .next(decider()).on("FAILED").to(step2())
    .from(decider()).on("COMPLETED").to(step3())
    .end()
    .build();
}
```





# Execution parallèle

---

*SpringBatch* permet de configurer un job avec des exécutions parallèles.

```
<split id="split1" next="step4">
  <flow>
    <step id="step1" parent="s1" next="step2"/>
    <step id="step2" parent="s2"/>
  </flow>
  <flow>
    <step id="step3" parent="s3"/>
  </flow>
</split>
<step id="step4" parent="s4"/>
```



# Java

---

```
@Bean
public Flow flow1() {
    return new FlowBuilder<SimpleFlow>("flow1")
        .start(step1())
        .next(step2()).build();
}
@Bean
public Flow flow2() {
    return new FlowBuilder<SimpleFlow>("flow2")
        .start(step3()).build();
}
@Bean
public Job job(Flow flow1, Flow flow2) {
    return this.jobBuilderFactory.get("job")
        .start(flow1)
        .split(new SimpleAsyncTaskExecutor())
        .add(flow2)
        .next(step4())
        .end().build();
}
```



# ItemReader / ItemWriter

---

## **Introduction**

Fichiers à plat

XML

JSON

Base de données



# Introduction

---

un ***ItemReader*** est le moyen de fournir des données provenant de nombreux types d'entrée

- Fichier plat : Lecture ligne par ligne de fichier.  
Chaque ligne contient un enregistrement divisé en champs à position fixe
- XML : Permet la validation vis à vis d'un schéma xsd
- BD : *ResultSet* mappé vers des objets (*RowMapper*)
- ...



# Interface

---

```
public interface ItemReader<T> {  
    T read() throws Exception, UnexpectedInputException,  
        ParseException, NonTransientResourceException;  
}
```

La méthode ***read ()*** définit le contrat le plus essentiel de ItemReader.

L'appel renvoie un élément (une ligne, un élément XML, un enregistrement de BD) ou *null* s'il ne reste plus d'éléments.



# *ItemWriter*

---

***ItemWriter*** propose la fonctionnalité inverse d'un `ItemReader`

Il écrit des enregistrements dans un fichier à plat, un fichier XML, une base, une file de messages, ...

```
public interface ItemWriter<T> {  
    void write(List<? extends T> items) throws Exception;  
}
```



# *ItemStream*

---

En général, dans le cadre d'un job, les *ItemReader* et les *ItemWriter* doivent être ouverts, fermés et nécessitent un mécanisme de persistance de l'état.

L'interface ***ItemStream*** définit ce contrat

```
public interface ItemStream {  
    void open(ExecutionContext executionContext) throws ItemStreamException;  
    void update(ExecutionContext executionContext) throws ItemStreamException;  
    void close() throws ItemStreamException;  
}
```



# *ExecutionContext*

---

Les clients d'un *ItemReader* ou *ItemWriter* qui implémentent également *ItemStream* doivent appeler *open()* avant tout appel à *read()*, afin d'ouvrir des fichiers, des connexions bd, ...

Les données nécessaires pour *open()* et *update()* peuvent alors être récupérées de *ExecutionContext*





# ItemReader / ItemWriter

---

Introduction

**Fichiers à plat**

XML

JSON

Base de données



# Fichiers à plat

---

## 2 types de fichiers à plat

- Délimité. Les champs sont séparés par un délimiteur
- A taille fixe : Les champs occupent une taille fixe

Un **FieldSet** est l'abstraction de Spring Batch permettant de définir les champs du fichier

Les champs peuvent alors être accédés par leur nom ou leur index



# *FlatFileItemReader*

---

FlatFileItemReader, fournit les fonctionnalités de base pour lire et parser les fichiers à plat

Ses 2 dépendances les plus importantes sont :

- Resource (Spring coeur)

Ex :

```
Resource resource = new FileSystemResource("resources/trades.csv");
```

- LineMapper : Conversion d'une ligne en un objet



# Propriétés de *FlatFileItemReader*

---

**comments** (String []) : Préfixes de ligne indiquant les commentaires.

**encoding** (String) : Encodage de texte. Par défaut  
*Charset.defaultCharset* ().

**lineMapper** (*LineMapper*) : Convertit une chaîne en objet représentant l'élément.

**linesToSkip** (*int*) : Nombre de lignes à ignorer au haut du fichier.

**recordSeparatorPolicy** (*RecordSeparatorPolicy*) : Détermine les fins de ligne

**ressource** (*Ressource*) : La ressource à partir de laquelle lire.

**skippedLinesCallback** (*LineCallbackHandler*) : Interface permettant de traiter les lignes ignorées

**strict** (booléen) : En mode *strict*, le *Reader* lève une exception si l'entrée la ressource n'existe pas. Sinon, il trace le problème et continue.



# *LineMapper*

**LineMapper** (équivalent à RowMapper) convertit une ligne (String) en Objet

```
public interface LineMapper<T> {  
    T mapLine(String line, int lineNumber) throws Exception;  
}
```

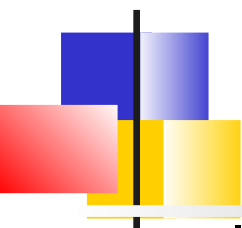
En entrée, il reçoit une ligne brute et il peut s'appuyer sur :

- **LineTokenizer** : Conversion d'une ligne en un ensemble de champs

```
public interface LineTokenizer {  
    FieldSet tokenize(String line);  
}
```

- **FieldSetMapper** : Conversion d'un FieldSet en un Objet du domaine

```
public interface FieldSetMapper<T> {  
    T mapFieldSet(FieldSet fieldSet) throws BindException;  
}
```



# Quelques implémentations

---

Implémentations de *LineTokenizer* :

- ***DelimitedLineTokenizer***:
- ***FixedLengthTokenizer***
- ***PatternMatchingCompositeLineTokenizer***

Implémentation de *LineMapper* : ***DefaultLineMapper***

```
public class DefaultLineMapper<T> implements LineMapper<>, InitializingBean
{
    public T mapLine(String line, int lineNumber) throws Exception {
        return fieldSetMapper.mapFieldSet(tokenizer.tokenize(line));
    }
}
```

***BeanWrapperFieldSetMapper*** : Associe automatiquement les champs avec les noms de propriétés d'un JavaBean



# *FixedLengthLineTokenizer*

---

@Bean

```
public FixedLengthTokenizer fixedLengthTokenizer() {  
    FixedLengthTokenizer tokenizer = new FixedLengthTokenizer();  
    tokenizer.setNames("ISIN", "Quantity", "Price", "Customer");  
    tokenizer.setColumns(new Range(1, 12), new Range(13, 15), new  
        Range(16, 20), new Range(21, 29));  
  
    return tokenizer;  
}
```



# *Exceptions*

---

2 types d'Exceptions :

- ***FlatFileParseException*** : Erreur à la lecture du fichier  
En général, Il n'y a rien faire si ce n'est échouer
- ***FlatFileFormatException*** : Erreur à la tokenization  
(*IncorrectTokenCountException*,  
*IncorrectLineLengthException*)  
Pour ce genre d'exception en général, on trace et on ignore la ligne





# *FlatFileItemWriter*

---

***FlatFileItemWriter*** permet d'écrire dans des fichiers délimités ou à taille fixe de manière transactionnelle.

Propriétés de *FlatFileItemWriter* :

- LineSeparator
- Encoding
- append, shouldDeleteIfExists
- HeaderCallback, FooterCallback



# *LineAggregator*

---

*FlatItemFileWriter* se base sur un ***LineAggregator***  
`write(lineAggregator.aggregate(item) + LINE_SEPARATOR);`

*LineAggregator* est le pendant de *LineTokenizer* :  
transforme un élément en une String

```
public interface LineAggregator<T> {  
    public String aggregate(T item);  
}
```

Implémentation basique par  
*PassThroughLineAggregator*<T> :  
`return item.toString();`



# Exemple Configuration Java

---

@Bean

```
public FlatFileItemWriter itemWriter() {  
    return new FlatFileItemWriterBuilder<Foo>()  
        .name("itemWriter")  
        .resource(new FileSystemResource("output.txt"))  
        .lineAggregator(new PassThroughLineAggregator<>())  
        .build();  
}
```



# *FieldExtractor*

---

Pour la conversion d'objet en ligne, *SpringBatch* propose la séquence suivante :

- Convertir les champs de l'élément en un tableau.
- Agréger le tableau en une ligne

Il propose alors l'interface ***FieldExtractor*** (équivalent de *FieldSetMapper*)

```
public interface FieldExtractor<T> {  
    Object[] extract(T item);  
}
```



# Quelques implémentations

---

## Implémentations *LineAggregator* :

- ***DelimitedLineAggregator***

```
DelimitedLineAggregator<CustomerCredit> lineAggregator = new  
DelimitedLineAggregator<>();  
lineAggregator.setDelimiter(",");  
lineAggregator.setFieldExtractor(fieldExtractor);
```

- ***FormatterLineAggregator***

```
FormatterLineAggregator<CustomerCredit> lineAggregator = new  
FormatterLineAggregator<>();  
lineAggregator.setFormat("%-9s%-2.0f");  
lineAggregator.setFieldExtractor(fieldExtractor);
```

## Implémentations *FieldExtractor* :

PassThroughFieldExtractor, BeanWrapperFieldExtractor



# Exemple configuration Java

---

```
// Fichier à taille fixe en utilisant un BeanWrapperFieldExtractor
```

```
@Bean
```

```
public FlatFileItemWriter<CustomerCredit> itemWriter(Resource  
    outputResource) throws Exception {
```

```
    return new FlatFileItemWriterBuilder<CustomerCredit>()  
        .name("customerCreditWriter")  
        .resource(outputResource)  
        .formatted()  
        .format("%-9s%-2.0f")  
        .names(new String[] {"name", "credit"})  
        .build();
```

```
}
```



# ItemReader / ItemWriter

---

Introduction  
Fichiers à plat  
**XML**  
JSON  
Base de données



# Introduction

---

Chaque élément correspond à un fragment XML

Fragment 1

Fragment 2

Fragment 3

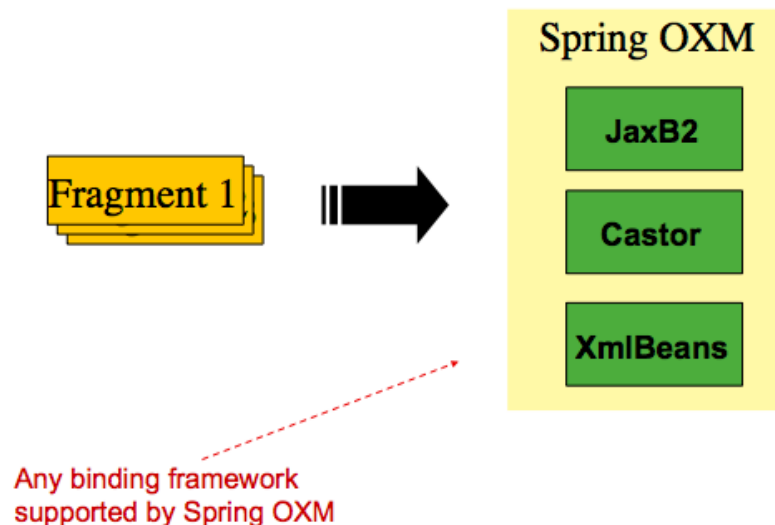
```
<trade>
  <isin>XYZ0001</isin>
  <quantity>5</quantity>
  <price>11.39</price>
  <customer>Customer1</customer>
</trade>
<trade>
  <isin>XYZ0002</isin>
  <quantity>2</quantity>
  <price>72.99</price>
  <customer>Customer2c</customer>
</trade>
<trade>
  <isin>XYZ0003</isin>
  <quantity>9</quantity>
  <price>99.99</price>
  <customer>Customer3</customer>
</trade>
```





# Spring OXM

Les fragments XML sont convertis en objet via l'interface **Spring OXM** qui supporte plusieurs implémentations



Le parsing est effectué en mode flux (*StAX API*)



# *StaxEventItemReader*

---

***StaxEventItemReader*** permet de traiter un fichier d'entrée en XML.

Sa configuration consiste en :

- Fournir le nom de l'**élément racine** identifiant un élément dans le XML
- Le fichier **resource**
- Un **Unmarshaller** permettant la conversion en objet  
*Object unmarshal(Source source)*



# *XstreamMarshaller*

---

XstreamMarshaller est une implémentation commune de Unmarshaller.

Il se configure avec qui accepte une map dont :

- la première clé/valeur est l'élément racine et le type d'objet à créer.
- Les autres clés valeurs correspondent au nom des autres éléments et aux types des attributs de l'objet.



# Configuration

---

@Bean

```
public XStreamMarshaller customerCreditMarshaller() {  
    XStreamMarshaller marshaller = new XStreamMarshaller();  
    Map<String, Class> aliases = new HashMap<>();  
    aliases.put("trade", Trade.class);  
    aliases.put("price", BigDecimal.class);  
    aliases.put("isin", String.class);  
    aliases.put("customer", String.class);  
    aliases.put("quantity", Long.class);  
    marshaller.setAliases(aliases);  
    return marshaller;  
}
```



# *Jaxb2Marshaller*

---

Avec ***Jaxb2Marshaller***, la configuration du marshallage s'effectue en annotant la classe du domaine avec les annotations JAXB

```
<bean id="reportUnmarshaller"
      class="org.springframework.xml.jaxb.Jaxb2Marshaller">
  <property name="classesToBeBound">
    <list>
      <value>com.mkyong.model.Report</value>
    </list>
  </property>
</bean>
```



# *Jaxb2Marshaller (2)*

---

```
@XmlElement(name = "record")
public class Report {
    @XmlAttribute(name = "refId")
    private int refId;
    @XmlElement
    private String name;
    @XmlElement(name = "age")
    private int age;
    @XmlJavaTypeAdapter(JaxbDateAdapter.class)
    @XmlElement
    private Date dob;
    @XmlJavaTypeAdapter(JaxbBigDecimalAdapter.class)
    @XmlElement
    private BigDecimal income;
```



# StaxEventItemWriter

---

La sortie fonctionne symétriquement à l'entrée.

Le StaxEventItemWriter a besoin d'une ressource, d'un marshaller et un élément racine.



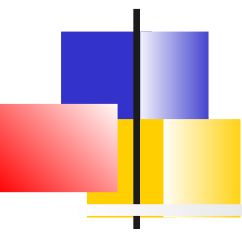
# Configuration

---

@Bean

```
public StaxEventItemWriter itemWriter(Resource outputResource) {  
    return new StaxEventItemWriterBuilder<Trade>()  
        .name("tradesWriter")  
        .marshaller(tradeMarshaller())  
        .resource(outputResource)  
        .rootTagName("trade")  
        .overwriteOutput(true)  
        .build();  
}
```





# ItemReader / ItemWriter

---

Introduction  
Fichiers à plat  
XML  
**JSON**  
Base de données



# Introduction

---

SpringBatch suppose que la ressource JSON est un tableau d'objets JSON correspondant à des éléments individuels.

Spring Batch n'est lié à aucune bibliothèque JSON particulière

```
[  
{  
  "isin": "123",  
  "price": 1.2,  
  "customer": "foo"  
},  
{  
  "isin": "456",  
  "price": 1.4,  
  "customer": "bar"  
}  
]
```



# *JsonItemReader*

---

***JsonItemReader*** délègue le parsing et le mapping à des implémentations de *JsonObjectReader*.

Cette interface est destinée à être implémentée à l'aide d'une API de streaming pour lire les objets JSON par blocs.

Deux implémentations sont actuellement fournies:

- Jackson : ***JacksonJsonObjectReader***
- Gson : ***GsonJsonObjectReader***



# Configuration

---

@Bean

```
public JsonItemReader<Trade> jsonItemReader() {  
    return new JsonItemReaderBuilder<Trade>()  
        .jsonObjectReader(new  
            JacksonJsonObjectReader<>(Trade.class))  
        .resource(new ClassPathResource("trades.json"))  
        .name("tradeJsonItemReader")  
        .build();  
}
```



# *JsonFileItemWriter*

---

***JsonFileItemWriter*** délègue le  
marshalling des éléments à  
***JsonObjectMarshaller***.

Interface responsable de générer le JSON à  
partir d'un objet.

2 implémentations :

- Jackson : ***JacksonJsonObjectMarshaller***
- Gson : ***GsonJsonObjectMarshaller***



# Support Multi-fichiers

---

Quelque soit le type (plat, xml, json), il est possible de traiter plusieurs fichiers en entrée qui ont le format

```
<bean id="multiResourceReader"  
  class="org.spr...MultiResourceItemReader">  
  <property name="resources" value="classpath:data/file-*.txt" />  
  <property name="delegate" ref="flatFileItemReader" />  
</bean>
```



# ItemReader / ItemWriter

---

Introduction  
Fichiers à plat  
XML  
JSON  
**Base de données**



# Introduction

---

Lors de l'utilisation d'une BD, une des problématique à éviter est de charger en mémoire l'ensemble des enregistrements d'une table généralement volumineuse.

*SpringBatch* offre 2 alternatives pour les `ItemReader` :

- Basé sur un curseur. (`ResultSet`)  
Lecture d'un item puis `next()`, les anciens items peuvent être désalloués
- Basé sur des pages. (`start + offset`)





# RowMapper

---

Les implémentations de ***RowMapper***<sup>1</sup> permettent de faire la transformation d'un enregistrement du *ResultSet* en un objet du domaine.

```
public CustomerCredit mapRow(ResultSet rs, int rowNum) throws
    SQLException {
    CustomerCredit customerCredit = new CustomerCredit();
    customerCredit.setId(rs.getInt(ID_COLUMN));
    customerCredit.setName(rs.getString(NAME_COLUMN));
    customerCredit.setCredit(rs.getBigDecimal(CREDIT_COLUMN));
    return customerCredit;
}
```

1. Cette interface est utilisée également par *JdbcTemplate*



# *JdbcCursorItemReader*

---

***JdbcCursorItemReader*** est l'implémentation JDBC basée sur le curseur.

Il fonctionne directement avec un ResultSet et nécessite une instruction SQL pour s'exécuter sur une connexion obtenue à partir d'un DataSource.



# Configuration

---

@Bean

```
public JdbcCursorItemReader<CustomerCredit> itemReader() {  
  
    return new JdbcCursorItemReaderBuilder<CustomerCredit>()  
        .dataSource(this.dataSource)  
        .name("creditReader")  
        .sql("select ID, NAME, CREDIT from CUSTOMER")  
        .rowMapper(new CustomerCreditRowMapper())  
        .build();  
}
```



# Propriétés de *JdbcCursorItemReader*

---

***ignoreWarnings*** (true) : Warning SQL

***maxRows*** : limite sur le maximum de ligne

***queryTimeout*** : nombre de secondes pour la  
requête

***verifyCursorPosition*** : Vérifier que personne à  
part le Reader n'a appelé *next()* sur le  
*ResultSet*

***saveState*** : L'état du reader est stocké dans le  
contexte d'exécution



# Hibernate

---

Hibernate n'est pas réputé pour être adapté au traitement batch. L'usage par défaut de la session garde les objets lu en mémoire

SpringBatch permet l'utilisation d'Hibernate en mode batch en utilisant une ***StatelessSession*** plutôt que la session par défaut

- Une *StatelessSession* enlève les fonctionnalités de cache de 1<sup>er</sup> niveau d'Hibernate



# *HibernateCursorItemReader*

***HibernateCursorItemReader*** permet de déclarer une instruction HQL et une *SessionFactory*.

Le mapping objet est effectué par les annotations Hibernate/JPA

@Bean

```
public HibernateCursorItemReader itemReader(SessionFactory sessionFactory) {  
    return new HibernateCursorItemReaderBuilder<CustomerCredit>()  
        .name("creditReader")  
        .sessionFactory(sessionFactory)  
        .queryString("from CustomerCredit")  
        .build();  
}
```



# *StoredProcedureItemReader*

***StoredProcedureItemReader*** fonctionne comme *JdbcCursorItemReader*, sauf qu'il faut lui fournir une procédure stockée qui renvoie un curseur.

@Bean

```
public StoredProcedureItemReader reader(DataSource dataSource) {  
    StoredProcedureItemReader reader = new StoredProcedureItemReader();  
    reader.setDataSource(dataSource);  
    reader.setProcedureName("sp_customer_credit");  
    reader.setRowMapper(new CustomerCreditRowMapper());  
    return reader;  
}
```



# JdbcPagingItemReader

---

Le ***JdbcPagingItemReader*** nécessite un ***PagingQueryProvider*** chargé de fournir les requêtes SQL utilisées pour récupérer les lignes d'une page.

Le ***SqlPagingQueryProviderFactoryBean*** permet de s'affranchir des spécificités de la base pour implémenter la pagination





# Configuration

---

*SqlPagingQueryProviderFactoryBean*  
nécessite de préciser :

- Une clause ***select***
- Une clause ***from***
- Une clause optionnelle ***where***
- Le ***sortkey*** (contrainte d'unicité dans la base)



# Exemple

---

@Bean

```
public JdbcPagingItemReader itemReader(DataSource dataSource, PagingQueryProvider
    queryProvider) {
    Map<String, Object> parameterValues = new HashMap<>();
    parameterValues.put("status", "NEW");
    return new JdbcPagingItemReaderBuilder<CustomerCredit>()
        .name("creditReader").dataSource(dataSource)
        .queryProvider(queryProvider).parameterValues(parameterValues)
        .rowMapper(customerCreditMapper())
        .pageSize(1000).build();
}
```

@Bean

```
public SqlPagingQueryProviderFactoryBean queryProvider() {
    SqlPagingQueryProviderFactoryBean provider = new SqlPagingQueryProviderFactoryBean();
    provider.setSelectClause("select id, name, credit");
    provider.setFromClause("from customer");
    provider.setWhereClause("where status=:status");
    provider.setSortKey("id");
    return provider;
}
```



# *JpaPagingItemReader*

---

Il est possible de faire de la pagination avec JPA (donc Hibernate)

@Bean

```
public JpaPagingItemReader itemReader() {  
    return new JpaPagingItemReaderBuilder<CustomerCredit>()  
        .name("creditReader")  
        .entityManagerFactory(entityManagerFactory())  
        .queryString("select c from CustomerCredit c")  
        .pageSize(1000)  
        .build();  
}
```



# *ItemWriter* pour les BDs

---

Il n'y a pas d'*ItemWriter* spécifique pour les bases, car les bases apportent déjà un comportement transactionnel

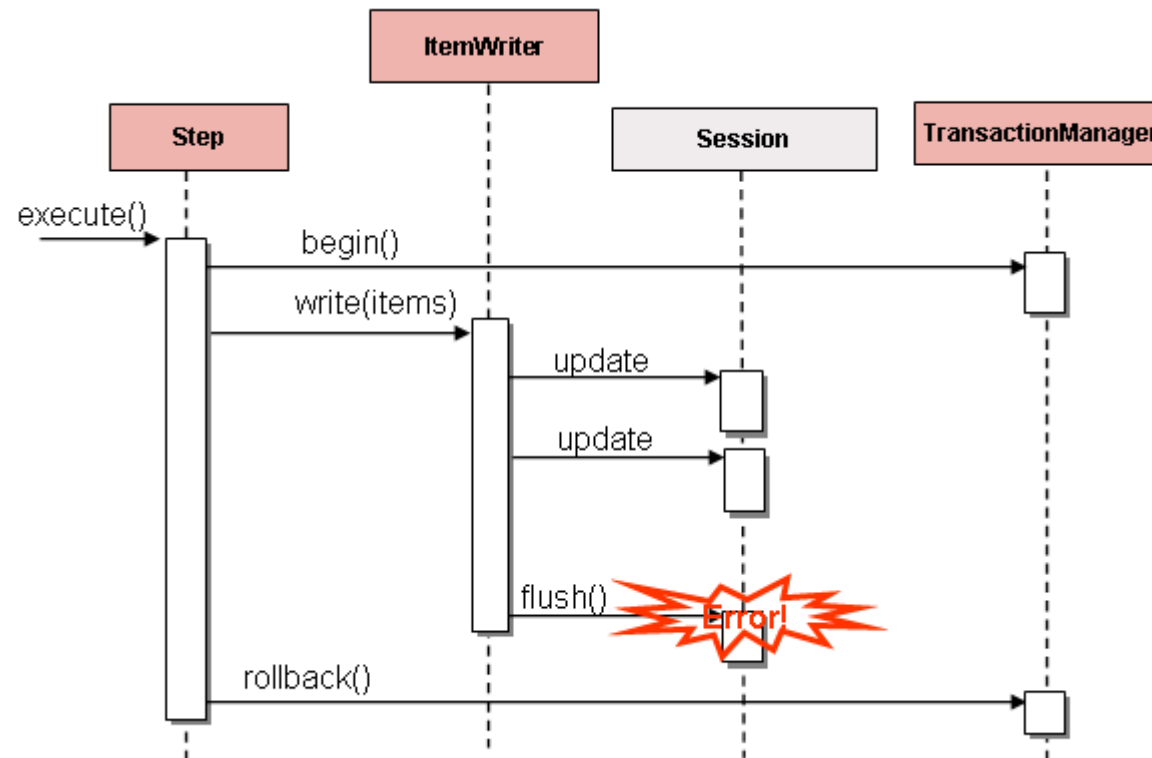
Il suffit de mettre au point ses propres DAO et d'implémenter l'interface de base *ItemWriter*

Cependant, 2 choses à surveiller dans un contexte de mise à jour par lot :

- les performances
- la gestion des erreurs

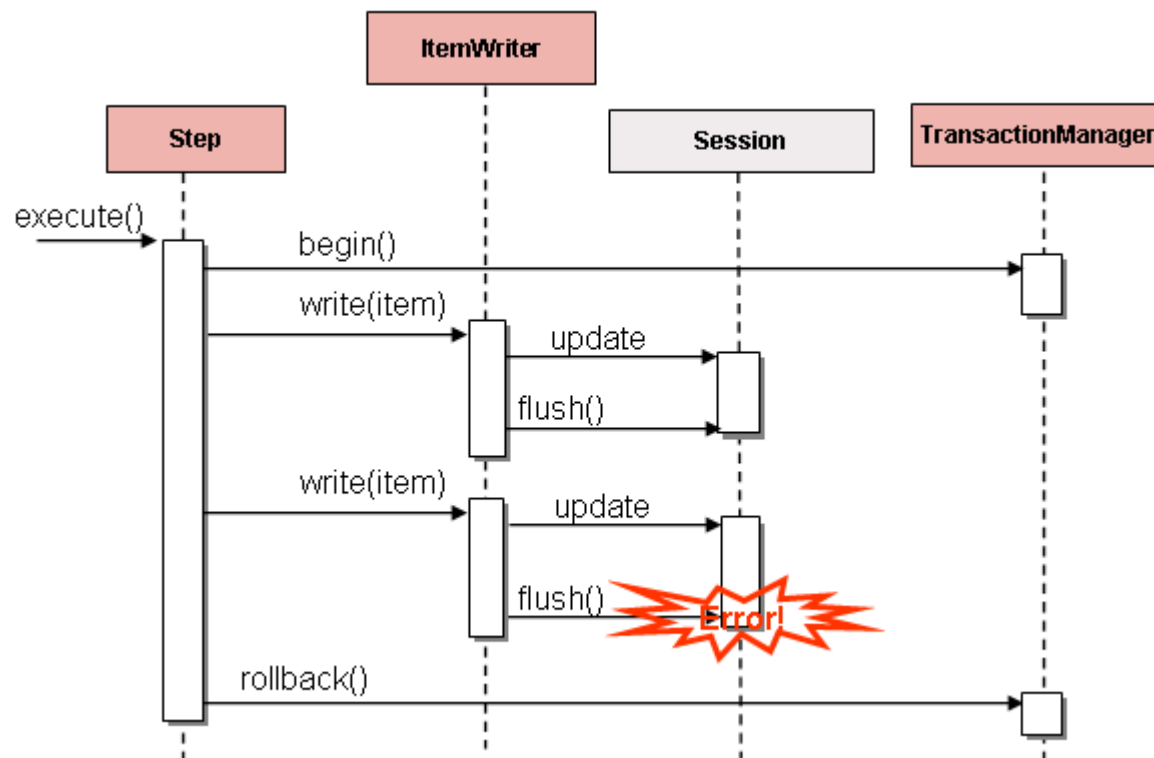
# Erreur et Batch

Toute erreur lors de l'écriture d'un nécessite un rollback complet du lot de mise à jour car il n'y a aucun moyen de savoir quel élément individuel a causé un exception



# Skip

Pour pouvoir skipper individuellement des éléments, il est nécessaire de flusher item par item





# ItemReader / ItemWriter

---

Introduction  
Fichiers à plat  
XML  
JSON  
Base de données  
**Compléments**



# Adapter

---

Spring Batch fournit des implémentations ***ItemReaderAdapter*** et ***ItemWriterAdapter*** permettant de réutiliser des classes existantes comme ItemReader et ItemWriter.

Pattern Adapter.





# Example

---

@Bean

```
public ItemReaderAdapter itemReader() {  
    ItemReaderAdapter reader = new ItemReaderAdapter();  
    reader.setTargetObject(fooService());  
    reader.setTargetMethod("generateFoo");  
    return reader;  
}
```

@Bean

```
public FooService fooService() {  
    return new FooService();  
}
```

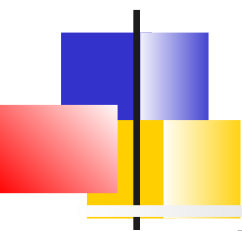


# Empêcher la persistance de l'état

---

Par défaut, les *ItemReader* et *ItemWriter* stockent leur état dans `ExecutionContext` avant un commit.

Cela peut être contrôlé par la propriété ***saveState***



# Redémarrage d'ItemReader

---

Lors de l'implémentation personnalisée d'un ItemReader, il faut prendre en compte les capacités de redémarrage.

Si l'on veut redémarrer le traitement à un endroit précis, il est nécessaire que l'ItemReader sauvegarde son état grâce à l'interface ItemStream



# Example

---

```
public class CustomItemReader<T> implements ItemReader<T>, ItemStream {
    List<T> items;
    int currentIndex = 0;
    private static final String CURRENT_INDEX = "current.index";
    public CustomItemReader(List<T> items) { this.items = items; }
    public T read() throws Exception {
        if (currentIndex < items.size()) { return items.get(currentIndex++); }
        return null;
    }
    public void open(ExecutionContext executionContext) throws ItemStreamException {
        if (executionContext.containsKey(CURRENT_INDEX)) {
            currentIndex = new Long(executionContext.getLong(CURRENT_INDEX)).intValue();
            } else { currentIndex = 0; }
    }
    public void update(ExecutionContext executionContext) throws ItemStreamException {
        executionContext.putLong(CURRENT_INDEX, new Long(currentIndex).longValue());
    }
    public void close() throws ItemStreamException {}
}
```



# Décorateurs

Spring Batch fournit des décorateurs prêts à l'emploi qui peuvent ajouter un comportement supplémentaire aux *ItemReader* et *ItemWriter*

- ***SynchronizedItemStreamReader*** / ***SynchronizedItemStreamWriter*** : Thread safe
- ***SingleItemPeekableItemReader*** : Permet une méthode peek qui lit un élément sans faire avancer le curseur
- ***MultiResourceItemWriter*** : Crée une nouvelle ressource de sortie tous les *itemCountLimitPerResource*
- ***ClassifierCompositeItemWriter*** : Permet d'avoir une Collection d'*ItemWriter*
- ***ClassifierCompositeItemProcessor*** : Permet d'avoir une Collection d'*ItemProcessor*



# Reader/Writer pour les messages brokers

---

SpringBatch fournit :

- ***AmqpItemReader / AmqpItemWriter*** :  
Utilise AmqpTemplate pour consommer ou produire des messages avec AMQP
- ***JmsItemReader / JmsItemWriter*** :  
Utilise JmsTemplate
- ***KafkaItemReader / KafkaItemWriter*** :  
Utilise KafkaTemplate

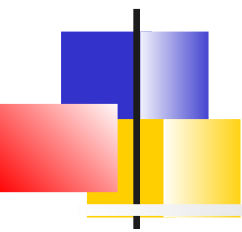


# Base de données

---

Spring Batch fournit :

- ***Neo4jItemReader, Neo4jItemWriter*** : Neo4j
- ***MongolItemReader, MongolItemWriter*** : MongoDB
- ***HibernateCursorItemReader, HibernatePagingItemReader, HibernateItemWriter*** : Hibernate
- ***RepositoryItemReader, RepositoryItemWriter*** :  
Spring Data
- ***JdbcBatchItemWriter*** : utilisation de  
NamedParameterJdbcTemplate
- ***JpaItemWriter***
- ***GemfireItemWriter*** : GemfireTemplate



# ItemReader / ItemWriter

---

Introduction  
Fichiers à plat  
XML  
JSON  
Base de données  
Compléments sur les readers/writers  
**ItemProcessor**





# Introduction

---

Spring Batch fournit l'interface ***ItemProcessor*** permettant de traiter (i.e transformer) un élément

```
public interface ItemProcessor<I, O> {  
    O process(I item) throws Exception;  
}
```

Un *ItemProcessor* peut être associé optionnellement à une Step

```
<step name="step1">  
    <tasklet>  
        <chunk reader="fooReader" processor="fooProcessor" writer="barWriter"  
            commit-interval="2"/>  
    </tasklet>  
</step>
```



# Chaînage

---

Les *ItemProcessor* peuvent être chaînée  
via ***CompositeItemProcessor***

*CompositeItemProcessor* prend en  
configuration une liste de  
*ItemProcessor*

Le traitement est délégué à chaque  
élément de la liste



# Configuration Java

---

```
@Bean
public Step step1() {
    return this.stepBuilderFactory.get("step1")
        .<Foo, Foobar>chunk(2)
        .reader(fooReader())
        .processor(compositeProcessor())
        .writer(fooBarWriter()).build();
}
```

```
@Bean
public CompositeItemProcessor compositeProcessor() {
    List<ItemProcessor> delegates = new ArrayList<>(2);
    delegates.add(new FooProcessor());
    delegates.add(new BarProcessor());
    CompositeItemProcessor processor = new CompositeItemProcessor();
    processor.setDelegates(delegates);
    return processor;
}
```



# Filterer les enregistrements

---

Pour filtrer un enregistrement, il suffit que l'ItemProcessor retourne null

Le framework évite alors d'ajouter cet élément à la liste des enregistrements livrés à ItemWriter.

Un exception levée par ItemProcessor entraîne un skip.



# Validation

---

Pour valider les éléments à traiter, Spring Batch fournit l'interface ***Validator***

```
public interface Validator<T> {  
    void validate(T value) throws ValidationException;  
}
```

On peut alors utiliser l'implémentation ***BeanValidatingItemProcessor*** qui s'appuie sur les annotations de la Bean Validation API (JSR-303)



# Pour aller plus loin

---

## **Scaling et traitement parallèle**

Répétition

Ré-essai

Tests unitaires

Patterns classiques



# Introduction

---

Pour augmenter les performances, on peut s'appuyer sur des traitements parallèles

SpringBatch permet 2 alternatives :

- Un unique processus multi-thread
- Plusieurs processus

Plus précisément :

- Une step multi-threadé d'un unique processus
- Des steps parallèles d'un unique processus
- Une step sur plusieurs process, les steps communiquant via un middleware
- Partitionnement d'une étape (unique ou multi-processus)



# Step multi-threadé

---

Il suffit d'ajouter un ***TaskExecutor***<sup>1</sup> à la configuration du Step

L'implémentation la plus simple est SimpleTaskExecutor qui démarre le traitement dans une thread séparé

- => Attention l'ordre des éléments n'est alors plus garantie
- => Attention, convient aux reader/writer stateless. La plupart de ceux fournis par SpringBatch sont stateful
- Par défaut, la tasklet limite le nombre de threads à 4

1. Fait partie de Spring Coeur, équivalent à Executor de Java





# Configuration

---

@Bean

```
public Step sampleStep(TaskExecutor taskExecutor) {  
    return this.stepBuilderFactory.get("sampleStep")  
        .<String, String>chunk(10)  
        .reader(itemReader())  
        .writer(itemWriter())  
        .taskExecutor(taskExecutor)  
        .throttleLimit(20)  
        .build();  
}
```



# Steps en parallèle

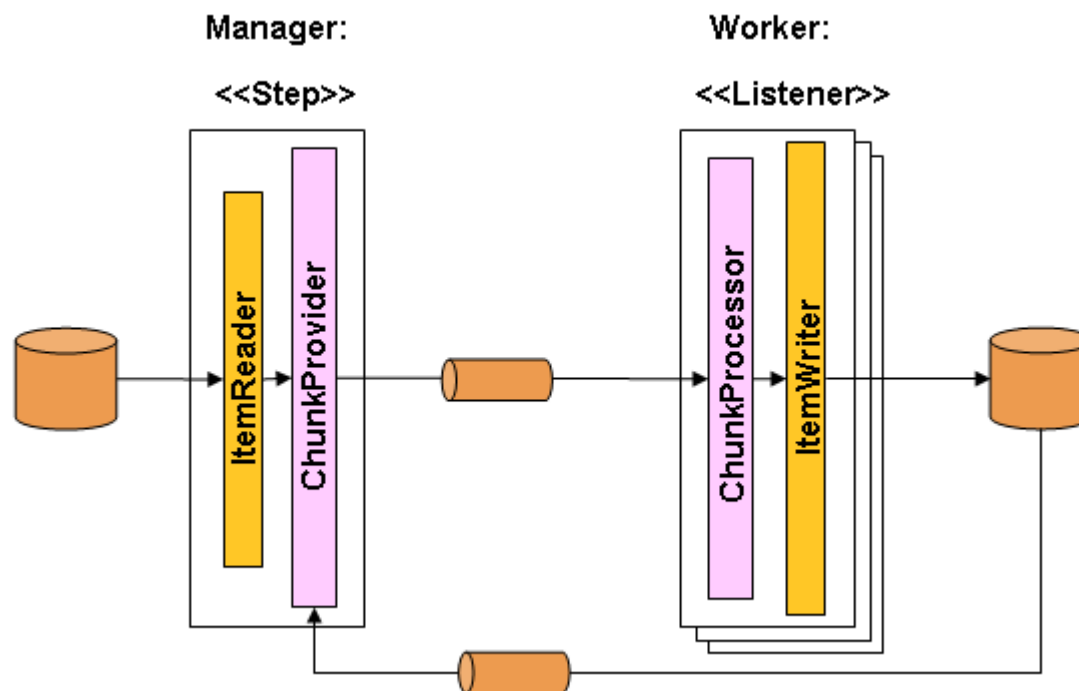
---

```
@Bean
public Flow flow1() {
    return new FlowBuilder<SimpleFlow>("flow1")
        .start(step1())
        .next(step2()).build();
}
@Bean
public Flow flow2() {
    return new FlowBuilder<SimpleFlow>("flow2")
        .start(step3()).build();
}
@Bean
public Job job(Flow flow1, Flow flow2) {
    return this.jobBuilderFactory.get("job")
        .start(flow1)
        .split(new SimpleAsyncTaskExecutor())
        .add(flow2)
        .next(step4())
        .end().build();
}
```

# Step séparé sur plusieurs processus

2 interfaces *ChunkProvider* et *ChunkProcessor*

Typiquement un producteur et un consommateur de message





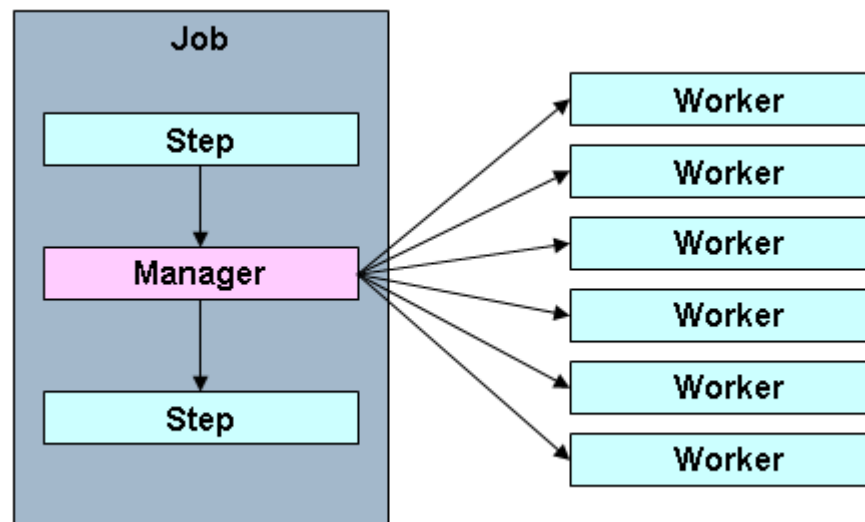
# Partitionnement

---

Un Step est un Manager

Les workers (locaux ou distants) exécutent la steps sous un sous-ensemble d'éléments , ils sont limités par l'attribut *grid-size*

Les métadonnées du *JobRepository* garantissent que chaque worker est exécuté une et une seule fois pour chaque exécution de Job





# *Partitioner*

---

***Partitioner*** est l'interface centrale pour créer des paramètres d'entrée pour une étape partitionnée sous la forme d'instances *ExecutionContext*.

- L'objectif est de créer un ensemble de valeurs d'entrée distinctes,  
par ex. un ensemble de plages de clés primaires , un ensemble de noms de fichiers uniques.

La méthode à implémenter est alors :

```
Map<String,ExecutionContext> partition(int gridSize)
```

- La clé de la Map contient le n° de partition
- Dans chaque ExecutionContext, on positionne les méta-données pour identifier le sous-ensemble des données à traiter pour cette partition



# Example

---

```
public class CustomMultiResourcePartitioner implements Partitioner {

    @Override
    public Map<String, ExecutionContext> partition(int gridSize) {
        Map<String, ExecutionContext> map = new HashMap<>(gridSize);
        int i = 0;
        for (Resource resource : resources) {
            ExecutionContext context = new ExecutionContext();
            Assert.state(resource.exists(), "Resource does not exist: "
                + resource);
            context.putString("fileName", resource.getFilename());
            context.putString("opFileName", "output"+i+".xml");
            map.put(PARTITION_KEY + i, context);
            i++;
        }
        return map;
    }
}
```



# Exemple (2)

---

```
// Création du bean et initialisation des fichiers à traiter
@Bean
public CustomMultiResourcePartitioner partitioner() {
    CustomMultiResourcePartitioner partitioner
        = new CustomMultiResourcePartitioner();

    Resource[] resources;
    try {
        resources = resourcePatternResolver
            .getResources("file:src/main/resources/input/*.csv");
    } catch (IOException e) {
        throw new RuntimeException("I/O ", e);
    }
    partitioner.setResources(resources);
    return partitioner;
}
```



# Configuration

---

```
/* La configuration définit la step principale et les step de type worker
Elle utilise également un taskExecutor pour que chaque worker
travaille dans sa Thread
Le nombre de partition est calé sur le nombre de threads */
@Bean
public Step partitionStep()
    throws UnexpectedInputException, MalformedURLException, ParseException {
    int gridSize=10 ;
    return steps.get("masterStep")
        .partitioner("workerStep", partitioner())
        .gridSize(gridSize)
        .step(workerStep())
        .taskExecutor(taskExecutor())
        .throttleLimit(gridSize)
        .build();
}
```





# Reader de workerStep

```
// Le nom du fichier du Reader est injecté au moment de la création de la step
@Bean
@StepScope
public FlatFileItemReader<Transaction>
    itemReader(@Value("#{stepExecutionContext[fileName]}") String filename)
        throws UnexpectedInputException, ParseException {
    FlatFileItemReader<Transaction> reader = new FlatFileItemReader<>();
    DelimitedLineTokenizer tokenizer = new DelimitedLineTokenizer();
    String[] tokens = {"username", "userid", "transactiondate", "amount"};
    tokenizer.setNames(tokens);
    reader.setResource(new ClassPathResource("input/partitioner/" + filename));
    DefaultLineMapper<Transaction> lineMapper = new DefaultLineMapper<>();
    lineMapper.setLineTokenizer(tokenizer);
    lineMapper.setFieldSetMapper(new RecordFieldSetMapper());
    reader.setLinesToSkip(1);
    reader.setLineMapper(lineMapper);
    return reader;
}
```



# Pour aller plus loin

---

Scaling et traitement parallèle

**Répétition**

Ré-essai

Tests unitaires

Patterns classiques



# Introduction

---

Le traitement par lots implique des actions répétitives, soit pour optimiser, soit dans d'un job

SpringBatch définit l'interface ***RepeatOperations***

```
public interface RepeatOperations {  
    RepeatStatus iterate(RepeatCallback callback)  
                        throws RepeatException;  
}
```

Le callback est également une interface :

```
public interface RepeatCallback {  
    RepeatStatus doInIteration(RepeatContext context)  
                        throws Exception;  
}
```



# Mécanisme

---

Le callback est exécuté à plusieurs reprises jusqu'à ce que l'implémentation détermine que l'itération doit se terminer.

La valeur de retour de ces interfaces est une énumération qui peut prendre :

- *RepeatStatus.CONTINUABLE*
- *RepeatStatus.FINISHED*



# Implémentation

---

L'implémentation générique de *RepeatOperations* est ***RepeatTemplate***

```
RepeatTemplate template = new RepeatTemplate();
template.setCompletionPolicy(new SimpleCompletionPolicy(2));
template.iterate(new RepeatCallback() {
    public RepeatStatus doInIteration(RepeatContext context) {
        // Traitement
        // RepeatContext peut être utilisé pour stocker des données
        // entre 2 appels
        return RepeatStatus.CONTINUABLE;
    }
});
```



# *CompletionPolicy*

---

La fin de la boucle dans la méthode itérative est déterminée par une ***CompletionPolicy***, qui est également une fabrique pour RepeatContext.

- Une fois que le callback a terminé doIteration, le *RepeatTemplate* appelle *CompletionPolicy* pour lui demander de mettre à jour son état (stocké dans *RepeatContext*).

Ensuite, il lui demande si l'itération est terminée

L'implémentation *SimpleCompletionPolicy* permet l'exécution un nombre fixe de fois



# Exception

---

Si une exception est lancée dans le callback, *RepeatTemplate* consulte un ***ExceptionHandler***, qui peut décider de relancer ou non l'exception

```
public interface ExceptionHandler {  
    void handleException(RepeatContext context,  
        Throwable throwable) throws Throwable;  
}
```



# Listener

---

*RepeatTemplate* permet d'enregistrer des ***RepeatListener***

```
public interface RepeatListener {  
    void before(RepeatContext context);  
    void after(RepeatContext context, RepeatStatus result);  
    void open(RepeatContext context);  
    void onError(RepeatContext context, Throwable e);  
    void close(RepeatContext context);  
}
```





# Pour aller plus loin

---

Scaling et traitement parallèle

Répétition

**Ré-essai**

Tests unitaires

Patterns classiques



# Introduction

---

Pour rendre le traitement plus robuste, il est parfois utile de réessayer automatiquement une opération qui a échoué au cas où elle réussirait lors d'une tentative ultérieure.

La fonctionnalité de retry a été retirée de Spring Batch à partir de la version 2.2.0. Il fait maintenant partie de la librairie Spring Retry.



# Interfaces

---

```
public interface RetryOperations {  
    <T, E extends Throwable> T execute(RetryCallback<T, E> retryCallback) throws E;  
    <T, E extends Throwable> T execute(RetryCallback<T, E> retryCallback,  
    RecoveryCallback<T> recoveryCallback) throws E;  
    <T, E extends Throwable> T execute(RetryCallback<T, E> retryCallback, RetryState  
        retryState) throws E, ExhaustedRetryException;  
    <T, E extends Throwable> T execute(RetryCallback<T, E> retryCallback,  
    RecoveryCallback<T> recoveryCallback, RetryState retryState) throws E;  
}
```

Et :

```
public interface RetryCallback<T, E extends Throwable> {  
    T doWithRetry(RetryContext context) throws E;  
}
```

Le callback s'exécute et, s'il échoue (en lançant une exception), il est retenté jusqu'à ce qu'il réussisse ou que l'implémentation abandonne.



# *RetryTemplate*

---

L'implémentation générique de  
*RetryOperation* est ***RetryTemplate***

```
RetryTemplate template = new RetryTemplate();
TimeoutRetryPolicy policy = new TimeoutRetryPolicy();
policy.setTimeout(30000L);
template.setRetryPolicy(policy);
Foo result = template.execute(new RetryCallback<Foo>() {
    public Foo doWithRetry(RetryContext context) {
        // Exécuter un traitement qui peut échouer
        return result;
    }
});
```



# *RecoveryCallback*

---

Lorsque les tentatives sont épuisées, les *RetryOperations* peuvent passer le contrôle à un autre callback, appelé ***RecoveryCallback***.

```
Foo foo = template.execute(new RetryCallback<Foo>() {  
    public Foo doWithRetry(RetryContext context) {  
  
    },  
    new RecoveryCallback<Foo>() {  
        Foo recover(RetryContext context) throws Exception {  
            // recovery  
        }  
    });
```



# *RetryPolicy*

---

La décision de réessayer ou d'échouer dans la méthode d'exécution est déterminée par un ***RetryPolicy***, qui est également une fabrique pour le *RetryContext*

Lors d'un échec du callback, *RetryTemplate* appelle *RetryPolicy* afin qu'il mette à jour son état (stocké dans le *RetryContext*) et qu'il décide si autre tentative peut être tentée.

- Si ce n'est pas possible, le *RetryPolicy* lance l'exception *RetryExhaustedException*,



# Implémentations

---

## Implémentation de RetryPolicy :

- ***SimpleRetryPolicy*** permet une nouvelle tentative en fonction
  - d'une liste d'Exception acceptée avec pour chaque d'un nombre fois
  - Une liste d'Exception fatales qui interdit de nouvelles tentatives
- ***ExceptionClassifierRetryPolicy*** permet une configuration plus fine que *SimpleRetryPolicy*
- ***TimeoutRetryPolicy*** : arrêt des tentatives après un timeout



# *BackoffPolicy*

---

Si un *RetryCallback* échoue, le *RetryTemplate* peut suspendre l'exécution selon le ***BackoffPolicy***

```
public interface BackoffPolicy {  
    BackOffContext start(RetryContext context);  
    void backOff(BackOffContext backOffContext)  
        throws BackOffInterruptedException;  
}
```

Une implémentation de *backOff()* consiste généralement à un appel à *Object.wait()*





# *Listener*

---

Spring Batch fournit également l'interface ***RetryListener*** qui permet d'être à l'écoute des tentatives

```
public interface RetryListener {  
    <T, E extends Throwable> boolean open(RetryContext context,  
        RetryCallback<T, E> callback);  
  
    <T, E extends Throwable> void onError(RetryContext context,  
        RetryCallback<T, E> callback, Throwable throwable);  
  
    <T, E extends Throwable> void close(RetryContext context,  
        RetryCallback<T, E> callback, Throwable throwable);  
}
```



# Pour aller plus loin

---

Scaling et traitement parallèle

Répétition

Ré-essai

**Tests unitaires**

Patterns classiques





