

Le traitement par lot avec Spring Batch

David THIBAU – 2025

david.thibau@gmail.com



Agenda

Introduction

- Batch processing
- Spring Batch : Architecture et Concepts
- Auto-configuration Spring Boot

Premiers Jobs avec SpringBatch

- Configuration basique d'un job
- Traitement des fichiers à plat
- ItemProcessor
- Traitement fichiers XML, JSON
- Intégration avec Base de données
- *JobScope* et *StepScope*
- Compléments

Configuration Jobs

- Démarrage
- Accès aux méta-données

Configuration Steps

- Traitement par morceau
- Redémarrage / Skip / Retry
- Listeners
- Tasklet
- StepFlow

Pour aller + loin

- Scaling et traitement parallèle
- Répétition et ré-essai
- Tests unitaires
- Patterns classiques



Introduction

Batch Processing

Spring Batch : Architecture et concepts
Autoconfiguration *Spring Boot*



Batch Processing

- ❖ De nombreuses applications d'entreprise nécessitent un **traitement en lots** :
 - Traitement automatisé de gros volumes d'informations sans intervention de l'utilisateur.
 - Application périodique de règles métier complexes sur de très grands ensembles de données, traitées de manière répétitive
 - Intégration des informations reçues des systèmes internes et externes qui nécessitent un formatage, une validation et un traitement transactionnel dans le système de persistance



Principe

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.

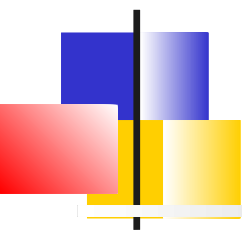


Scheduling

Les batchs ont souvent besoin d'être planifiés par un ***scheduler***.

SpringBatch n'est pas un *scheduler*, différentes alternatives existent :

- Scheduler système (*crontab* ou produit spécifique comme Tivoli, Control-M)
- Librairie Java Quartz
<http://www.quartz-scheduler.org/>
- Spring Task Scheduler et son annotation *@Scheduled* (basé sur Quartz)



Problématiques récurrentes

- Validez périodiquement le traitement (commit, rollback),
=> transactions par lots
- Traitement parallèle, massivement parallèle
- Traitement par étapes
 - Contraintes séquentielles, i.e. ~workflow
 - Reprise sur erreur, ignorer certains enregistrements lors d'une reprise
 - Réutilisation d'étapes dans différents batches



Recommandations (1)

- Simplifier autant que possible et éviter de construire des structures logiques complexes en un seul batch.
- Garder le traitement et le stockage des données physiquement proches
- Minimiser l'utilisation des ressources système, en particulier les I/O.
- Effectuer autant d'opérations que possible en mémoire.
- Allouer suffisamment de mémoire dès le départ pour éviter des réallocations en cours du processus.



Recommandations (2)

- Ne pas faire deux fois les mêmes traitements.
- Envisager le pire concernant l'intégrité des données.
=> Insérer des contrôles adéquats
- Utiliser des checksums pour la validation interne.

Par exemple, un enregistrement de fin indiquant le total des enregistrements

- Planifier et exécuter des tests de résistance au plus tôt avec des volumes de données conformes à la production.



Introduction

Batch Processing

***Spring Batch : Architecture et
concepts***

Autoconfiguration *Spring Boot*



Eco-système Spring

Spring Batch s'appuie sur ***Spring Framework***

:

- Productivité,
- Pattern IoC, approche de développement basée sur des POJOs
- Simplicité d'utilisation générale

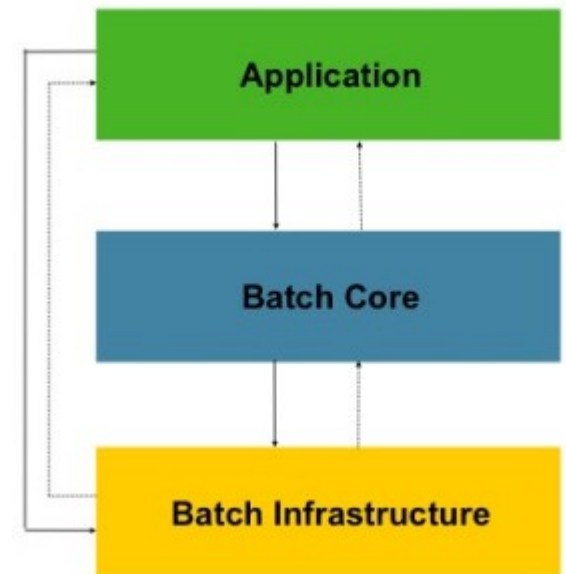
Il est également intégré sous forme de *starter* ***SpringBoot***.

Spring Batch est une des seules offres OpenSource qui fournit un framework robuste et scalable pour les traitements en lots.

Introduction

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 cœur et par l'application





API SpringBatch

L'API *Spring Batch* est constituée de :

- ***Jobs*** : Une application batch
- Eux-mêmes constitués d'étapes : les ***Steps***
- Elles-mêmes constitués d'unités de traitement :
ItemReader, ItemProcessor, ItemWriter et ***Tasklet***.



ItemReader/Writer disponibles

Fichiers : FlatFile, StaX XML, JSON

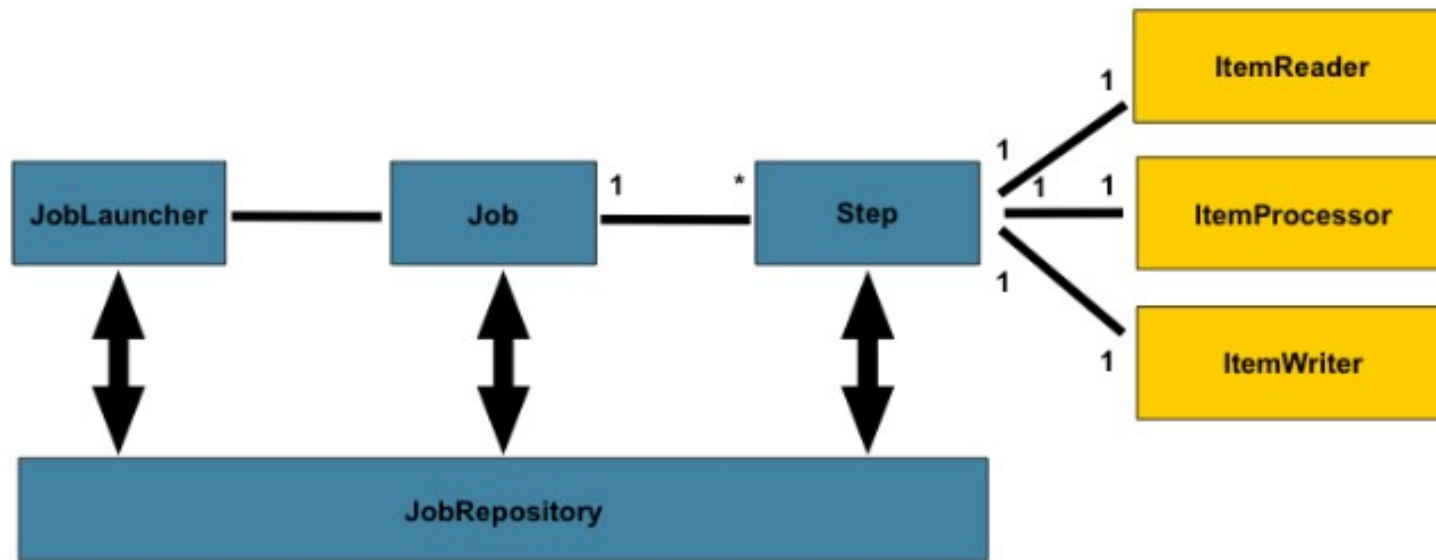
Message brokers : JMS, Amqp, Kafka

BD SQL : Jdbc, Hibernate, Jpa,
StoredProcedure,

NoSQL :Mongo, Neo4j

Code custom : Tasklet

Composants principaux





Job

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

- Configuré via XML ou Java.
- Combine plusieurs étapes (steps) qui appartiennent logiquement à un flux
- Configure des propriétés globales à toutes les étapes, propriétés de redémarrage par exemple



Configuration Java

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

Ex : *JobBuilder*

@Bean

```
public Job footballJob(JobRepository jobRepository) {  
    return new JobBuilder("footballJob", jobRepository)  
        .start(playerLoad())  
        .next(gameLoad())  
        .next(playerSummarization())  
        .build();  
}
```



Configuration XML

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

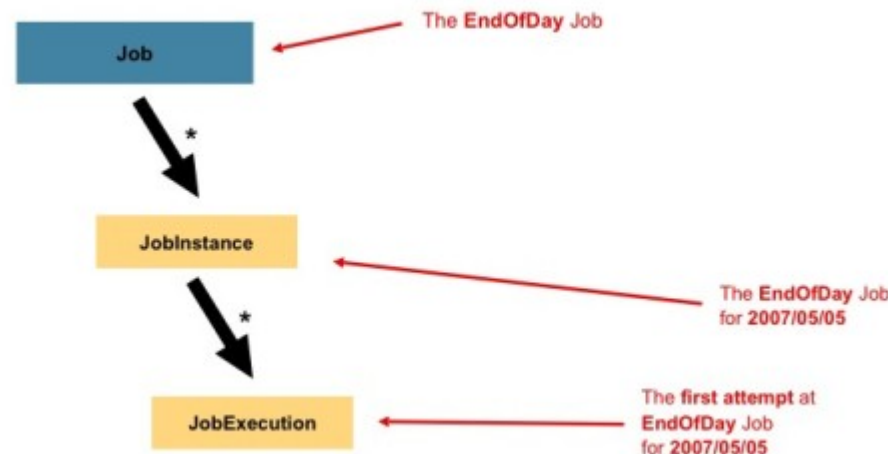
```
<job id="footballJob">  
  <step id="playerload" next="gameLoad"/>  
  <step id="gameLoad" next="playerSummarization"/>  
  <step id="playerSummarization"/>  
</job>
```

JobInstance

JobInstance représente un démarrage du job

Une seule instance d'un même job peut être démarrée à la fois

Un instance peut être associée à plusieurs exécutions ou tentatives (**JobExecution**), si certaines ont échoué





JobParameters

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

Les paramètres sont utilisés :

- Pour identifier une 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 de ses exécutions se termine avec succès.



Données persistantes

Certaines propriétés de *JobExecution* sont persistantes et stockées en BD:

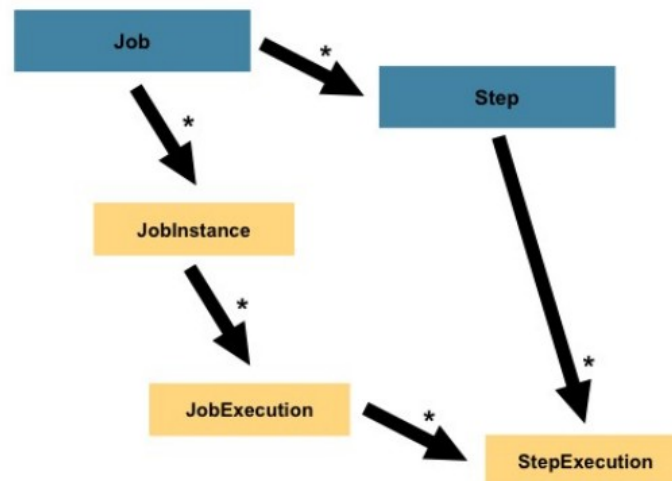
- **Status** (Énumération *BatchStatus*)
- **createTime**, **startTime**, **endTime**
- **ExitStatus** (Code de sortie système)
- **LastUpdated**
- **ExecutionContext** (Map de données spécifiques au job)
- **failureExceptions**

Step, StepExecution

Step encapsule une phase séquentielle indépendante d'un Job

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

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





StepExecution

StepExecution représente une tentative pour exécuter un *Step*.

Certaines de ses propriétés sont également persistantes :

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

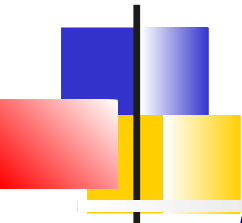


ExecutionContext

ExecutionContext représente une collection de paires clé / valeur

Permet de stocker l'état associé à un *StepExecution/JobExecution* pour :

- permettre un redémarrage
- produire des statistiques
- Transférer des données entres Steps
- ...
-



JobRepository

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

- Principalement utilisé par *JobLauncher*, *Job* et *Step*

Il est donc dépendant d'une datasource

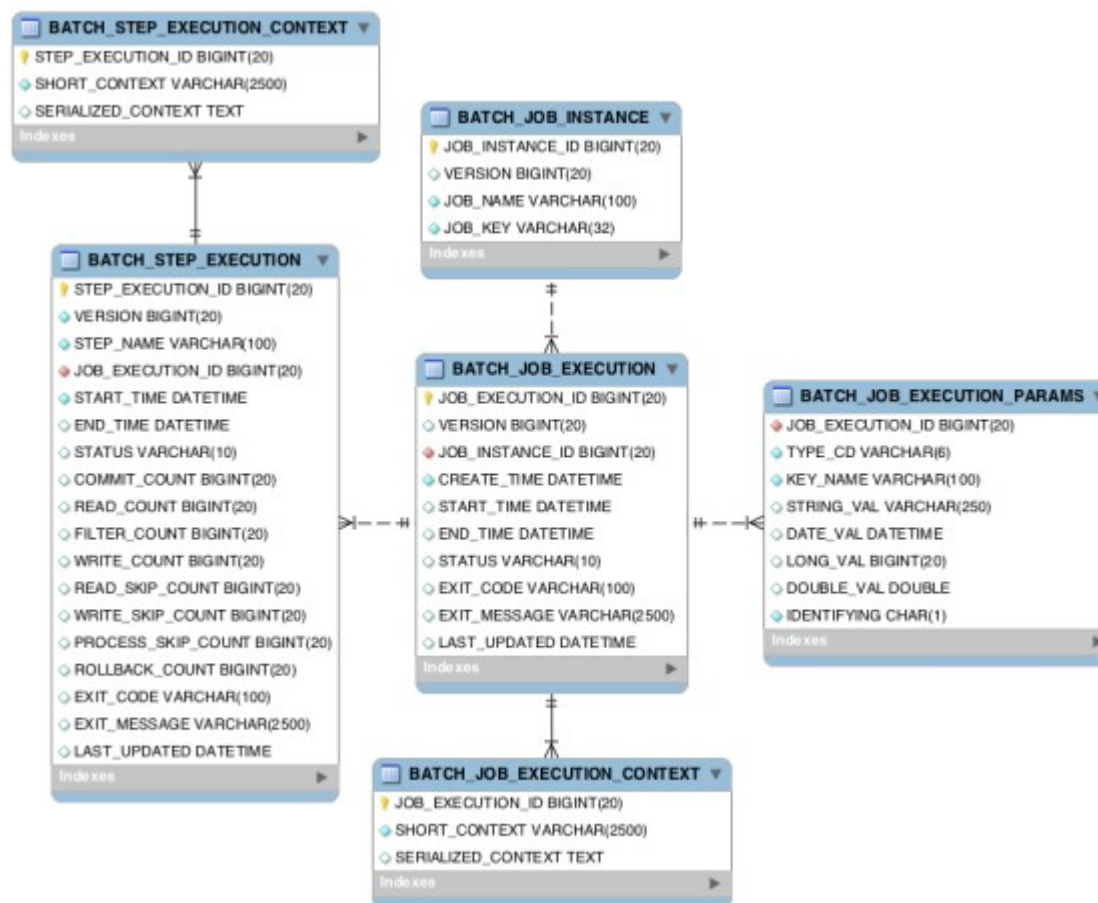
Configuration :

- XML par

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

- Java, configuration automatique via SpringBoot ou `@EnableBatchProcessing`

Schéma

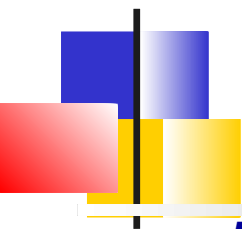




JobLauncher

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;  
  
}
```



Composants d'une de Step

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

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

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

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

ItemProcessor représente le traitement métier d'un élément.

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



Introduction

Batch Processing

Spring Batch : Architecture et concepts

Autoconfiguration *Spring Boot*



starter

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

Inclut :

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

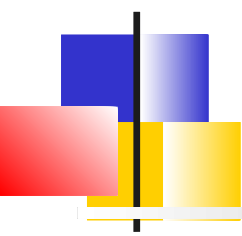


Activation configuration

Depuis la version 3.x de SpringBoot, la seule présence de la dépendance suffit pour activer l'auto-configuration de *SpringBatch*.

Autre moyens :

- Fournir un bean de type ***JobLauncher***
- Utiliser ***@EnableBatchProcessing***
(pour les versions + anciennes)



Beans de l'auto-configuration

`stepScope, jobScope`

`jobRepository`

`jobLauncher`

`jobRegistry,`

`jobExplorer`

`batchDataSourceInitializer`

`batchConfigurer`

`jobLauncherApplicationRunner`

`jobExecutionExitCodeGenerator`



Exécution de jobs

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

- Désactiver avec

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

- Les noms des Jobs à exécuter peuvent être fournis via:

```
spring.batch.job.names = job1, job2
```



Initialisation de la base

La propriété ***spring.batch.initialize-schema*** permet de contrôler si *SpringBatch* crée automatiquement les tables de la base.

Les valeurs possibles sont :

- *always*
- *embedeed*
- *never*

Les autres propriétés de *SpringBatch* concernent principalement le schéma de la base :

- *spring.batch.schema* : Chemin vers le script d'initialisation
- *spring.batch.table-prefix* : Préfixe des tables



Configuration custom

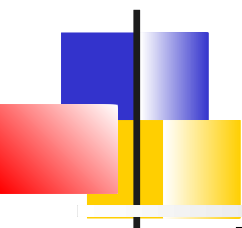
Pour implémenter ses propres beans SpringBatch, il suffit d'étendre ***DefaultBatchConfiguration*** et surcharger les méthodes qui nous intéressent

```
JobExplorer getJobExplorer()
```

```
JobLauncher getJobLauncher()
```

```
JobRepository getJobRepository()
```

```
PlatformTransactionManager getTransactionManager()
```



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`

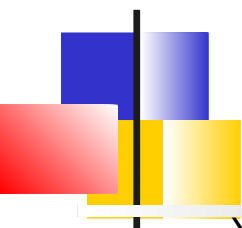


Configuration des transactions

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 que si on tente de démarrer en même temps 2 fois le même job. Un seul sera effectivement démarré.

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



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 DefaultBatchConfiguration

```
@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();  
}
```



Premier Jobs avec SpringBatch

Configuration basique d'un Job

Fichiers à plat

ItemProcessor

XML / JSON

Base de données

Compléments sur les Readers/Writers



Introduction

Considérations à prendre en compte, lors de la configuration :

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



Configuration basique

Configuration basique : Le job est une séquence de steps

Java

@Bean

```
public Job footballJob(JobRepository jobRepository) {  
    return new JobBuilder("footballJob", jobRepository)  
        .start(playerLoad())  
        .next(gameLoad())  
        .next(playerSummarization())  
        .build();  
}
```

XML

```
<job id="footballJob"> <!-- Utilise le bean jobRepository -->  
    <step id="playerload" parent="s1" next="gameLoad"/>  
    <step id="gameLoad" parent="s2" next="playerSummarization"/>  
    <step id="playerSummarization" parent="s3"/>  
</job>
```



Héritage d'un job avec XML

Même avec une configuration XML, on peut hériter d'une configuration parente¹ .

```
<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



Configuration basique d'une step

Java

@Bean

```
public Step playerLoad() {  
    return new StepBuilder("playerLoadStep", jobRepository)  
        .<Player, Player>chunk(10, transactionManager)  
        .reader(playerReader)  
        .writer(playerWriter)  
        .build();  
}
```

XML

```
<step id="step1">  
    <tasklet transaction-manager="transactionManager">  
        <chunk reader="playerReader" writer="playerWriter" commit-  
            interval="10"/>>  
        </tasklet>  
    </step>
```



Re-démarrage

Le lancement d'un Job est considéré comme un redémarrage si un *JobExecution* avec les **mêmes paramètres identifiants** existe déjà pour ce Job.

Par défaut, une exécution peut être redémarrée seulement si la dernière exécution ne s'est pas terminée normalement.

- Possibilité d'interdire les redémarrage.

```
new JobBuilder("footballJob", jobRepository)
    .preventRestart()
```

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



JobListener

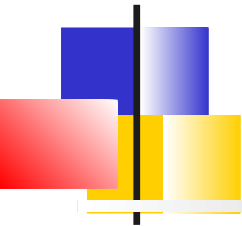
Des ***JobListeners*** peuvent être ajoutés à un Job permettant d'être à l'écoute des événements de démarrage et de fin.

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

Configuration :

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

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



Premier Jobs avec SpringBatch

Configuration basique d'un Job

Fichiers à plat

ItemProcessor

XML / JSON

Base de données

Compléments sur les Readers/Writers



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

FieldSet est l'abstraction de Spring Batch qui contient la définition des champs du fichier

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

```
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;  
}
```

L'implémentation par défaut **DefaultLineMapper** s'appuie sur 2 interfaces :

- **LineTokenizer** : Conversion d'une ligne en un *FieldSet*

```
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;  
}
```



Implémentations de LineTokenizer

DelimitedLineTokenizer: Séparation avec un caractère délimiteur.

FixedLengthTokenizer : A partir d'une taille fixe de caractère pour chaque champ

PatternMatchingCompositeLineTokenizer : Utilisé lorsque le format des lignes peut varier

Exemple FixedLengthTokenizer :

```
@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;
}
```



Implémentation *FieldSetMapper*

L'implémentation la plus courante et la plus simple est

BeanWrapperFieldSetMapper qui associe automatiquement les champs avec les propriétés d'un JavaBean



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



Exemple complet

```
@Bean
public FlatFileItemReader<Employee> reader() {
    FlatFileItemReader<Employee> reader = new FlatFileItemReader<Employee>();
    //Positionner la ressource d'entrée
    reader.setResource(new FileSystemResource("input/inputData.csv"));
    //Nombre de lignes d'entête à ignorer
    reader.setLinesToSkip(1);
    // LineMapper
    reader.setLineMapper(new DefaultLineMapper() {
        {
            //3 colonnes par ligne
            setLineTokenizer(new DelimitedLineTokenizer() { {
                setNames(new String[] { "id", "firstName", "lastName" });
            } });
            //Mapping automatique dans la classe Employee
            setFieldSetMapper(new BeanWrapperFieldSetMapper<Employee>() {
                { setTargetType(Employee.class); }
            });
        }
    });
    return reader;
}
```



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* :
Génération des entêtes ou bas de fichier



LineAggregator

FlatItemFileWriter se base sur un ***LineAggregator***

```
write(lineAggregator.aggregate(item) + LINE_SEPARATOR);
```

LineAggregator est le pendant de *LineTokenizer*, il transforme un objet 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();
}
```



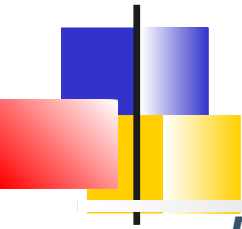
FieldExtractor

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

Le *LineAgregator* s'appuie alors sur l'interface ***FieldExtractor*¹**

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



Quelques implémentations de *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 : Retourne l'objet entier

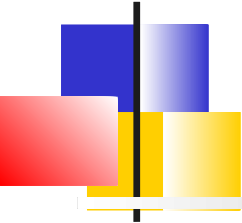
BeanWrapperFieldExtractor : Via un tableau de noms, il appelle les getters sur les items



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();
}
```



Premier Jobs avec SpringBatch

Configuration basique d'un Job

Fichiers à plat

ItemProcessor

XML / JSON

Base de données

Compléments sur les Readers/Writers



ItemProcessor

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 chaînés
via ***CompositeItemProcessor***

- *CompositeItemProcessor* encapsule une liste de *ItemProcessor*
- Le traitement est délégué à chaque élément de la liste



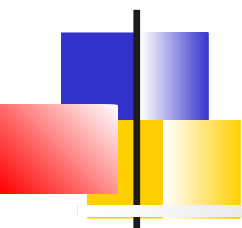
Configuration Java

@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 *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 l'arrêt du processus.



Validation

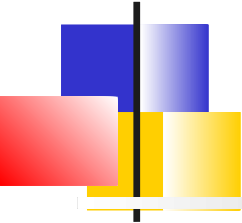
Pour valider les éléments à traiter, Spring Batch fournit la sous-interface

ValidatingItemProcessor

Ces processeurs valident les données sans les modifier. Ils s'appuient sur l'interface ***Validator***.

- Si le *Validator* lance une exception, 2 comportements en fonction de la propriété *filter* :
 - false, le process tombe en erreur
 - true : l'élément est ignoré

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



Premier Jobs avec *SpringBatch*

Configuration basique d'un Job

Fichiers à plat

ItemProcessor

XML / JSON

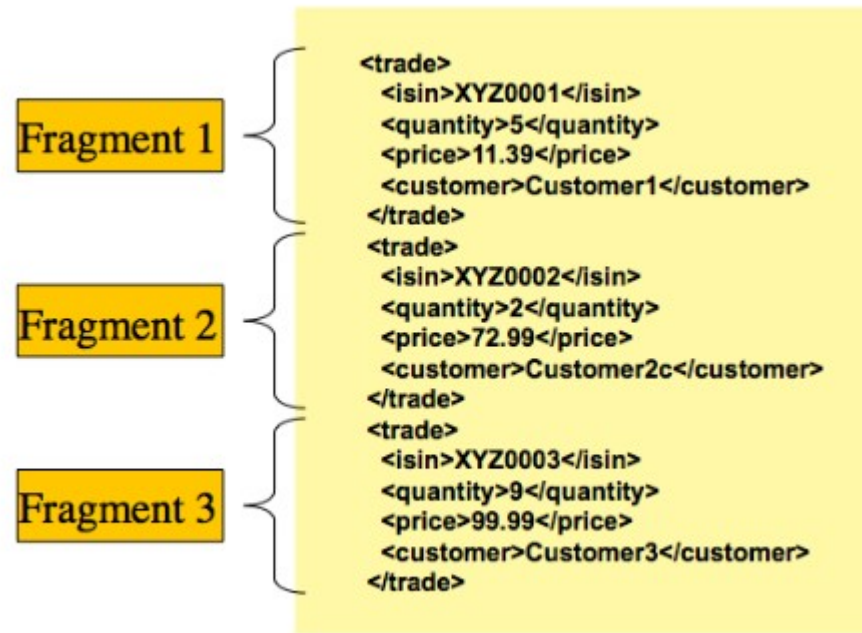
Base de données

Compléments sur les Readers/Writers



Introduction

Chaque élément correspond à un fragment XML

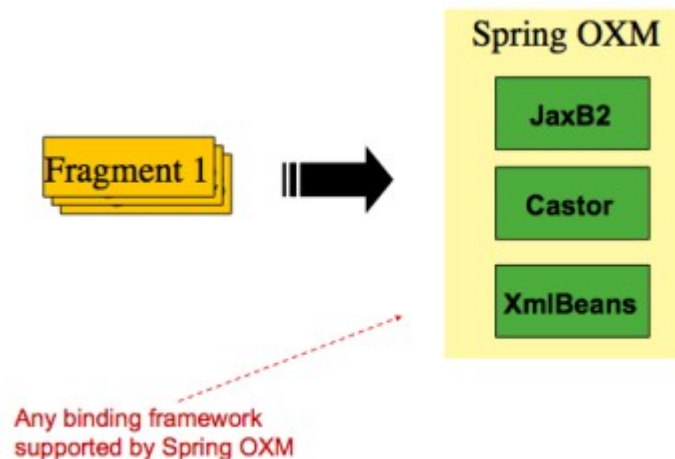




Spring OXM

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 **ressource**
- Un ***Unmarshaller*** permettant la conversion en objet
Object unmarshal(Source source)



XStreamMarshaller

XStreamMarshaller est une implémentation commune de *Unmarshaller*.

Il se configure avec 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();  
}
```



Ressource JSON

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***



Premier Jobs avec SpringBatch

Configuration basique d'un Job

Fichiers à plat

ItemProcessor

XML, JSON

Base de données

Compléments sur les Readers/Writers



Introduction

Lors de l'utilisation d'une BD, une des problématiques 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)

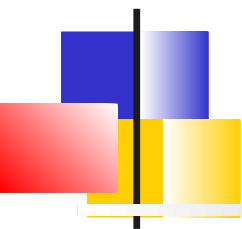


RowMapper

Les implémentations de **RowMapper**¹ 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*

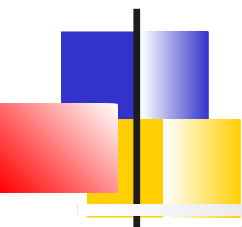
ignoreWarnings (true) : Warning SQL

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



JdbcPagingItemReader

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

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)



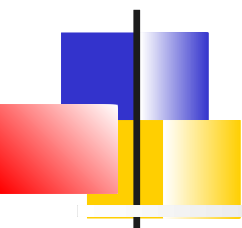
Example

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



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;  
}
```

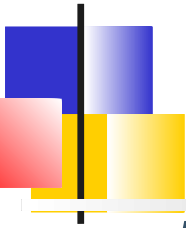


Hibernate

Hibernate n'est pas réputé pour être adapté au traitement batch. L'usage par défaut de la session garde les objets lus 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^{er} niveau d'Hibernate



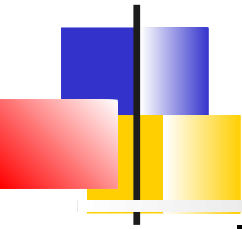
HibernateCursorItemReader

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();
}
```



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*

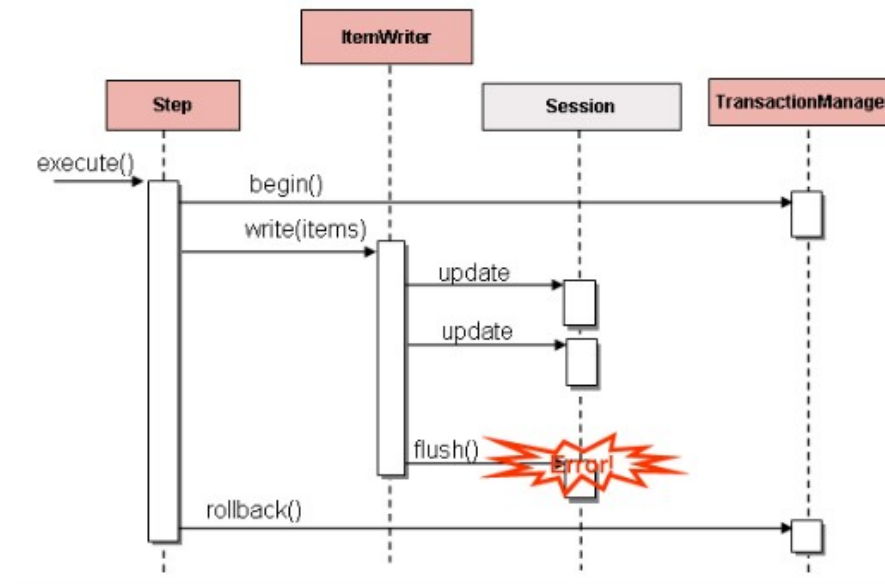
Cependant, 2 choses à surveiller dans un contexte de mise à jour par lot :

- les performances, cadence des commits, utilisation de pool de connexions
- la gestion des erreurs

Erreur et Batch

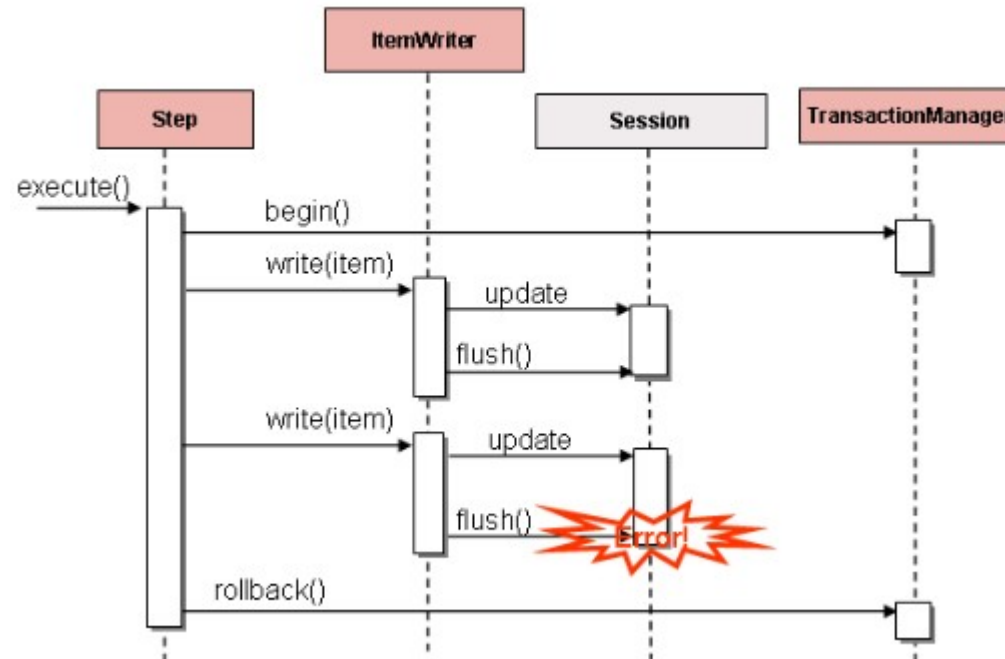
Toute erreur lors de l'écriture d'un lot nécessite un rollback complet du lot car il n'y a aucun moyen de savoir quel élément a causé une exception.

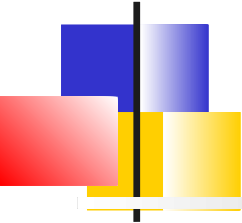
Exemple Hibernate, valable pour JDBC Batch



Skip

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





Premier Jobs avec *SpringBatch*

Configuration basique d'un Job

Fichiers à plat

ItemProcessor

XML, JSON

Base de données

**Compléments sur les
Readers/Writers**



Interface *ItemStream*

En général, dans le cadre d'un job, les *ItemReader* et les *ItemWriter* doivent ouvrir/fermer leur ressource et nécessitent un mécanisme pour stocker un état.

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

```
public interface ItemStream {  
    // Ouverture de la ressource à partir des informations du contexte  
    void open(ExecutionContext executionContext) throws ItemStreamException;  
    // permet de sauvegarder un état concernant la ressource  
    void update(ExecutionContext executionContext) throws ItemStreamException;  
    // Fermeture de la ressource  
    void close() throws ItemStreamException  
}
```

Les *ItemReader/Writer* fournis par SpringBatch implémentent cette interface



ExecutionContext

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

- => Les données nécessaires pour *open()* peuvent alors être récupérées de *ExecutionContext*
- => Les données nécessaires pour sauvegarder l'état sont écrit dans le context lors de la méthode *update()*

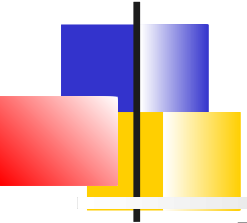
Dans le cadre d'une Step, *SpringBatch* crée un *ExecutionContext* pour chaque *StepExecution*, on peut alors y stocker un état pouvant être réutilisé lors d'un redémarrage



Enregistrement des *ItemStream*

Si l'interface *ItemStream* n'est pas implémentée par un *ItemReader/Writer* mais une classe déléguée, il faut absolument l'enregistrer via la balise ou méthode ***stream()***

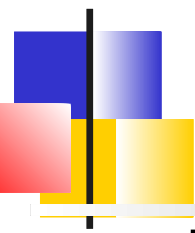
```
@Bean
public Step step1() {
    return this.stepBuilderFactory.get("step1")
        .<String, String>chunk(2)
        .reader(fooReader())
        .processor(fooProcessor())
        .writer(compositeItemWriter())
        .stream(barWriter())
        .build();
}
```



Empêcher la persistance de l'état

Par défaut, les *ItemReader* et *ItemWriter* stockent leurs états dans *ExecutionContext* avant un commit.

Cela peut être contrôlé par la propriété ***saveState*** présente dans tous les *readers* et *writers* fournis par SpringBatch



Implémentation 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 {}
}
```




Adaptateurs

Spring Batch fournit des implémentations ***ItemReaderAdapter*** et ***ItemWriterAdapter***¹ permettant de réutiliser des classes existantes comme *ItemReader* et *ItemWriter*.



Example

@Bean

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

@Bean

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



Décorateurs

Spring Batch fournit des décorateurs 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
- ***MultiResourceItemReader*** : Lit les items à partir de plusieurs ressources
- ***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



Example

```
<bean id="multiResourceReader"  
class=" org.springframework.batch.item.file.MultiResourceItemReader">  
<property name="resources" value="file:csv/inputs/domain-*.csv" />  
<property name="delegate" ref="flatFileItemReader" />  
</bean>
```

```
return new MultiResourceItemWriterBuilder<MyItem>()  
    .name("successReportItemWriter")  
    .itemCountLimitPerResource(1000)  
    .delegate(flatFileItemWriter())  
    .resource(new FileSystemResource("/output.csv"))  
    .resourceSuffixCreator(suffixCreator)  
    .build();
```



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



Jobs

Démarrage
Accès aux méta-données



Exécution d'un job

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

- Via une commande en ligne
- Via un scheduler
- Via une application Web
- ..

Cela consiste généralement à :

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



Démarrage

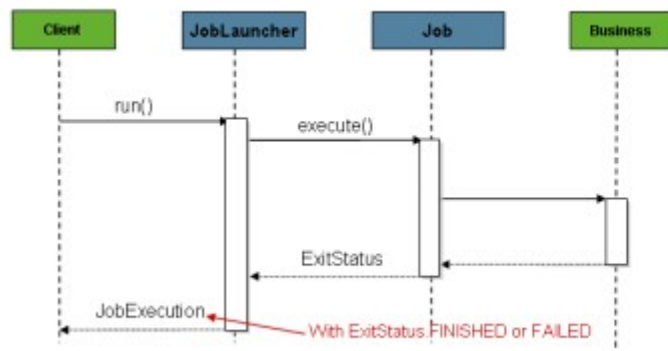
Le *JobLauncher* démarre un job grâce à sa méthode

```
JobExecution run(Job job, JobParameters jobParameters)
```

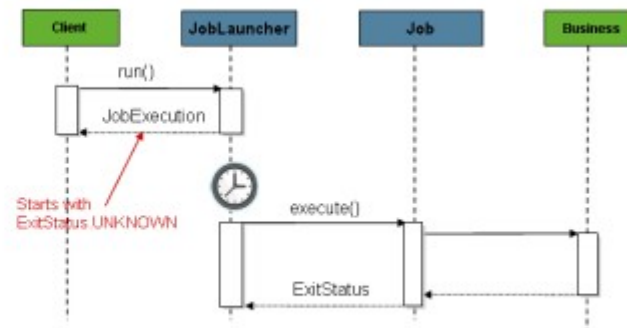
Le type de retour *jobExecution* contient les informations de l'exécution

Synchrone / Asynchrone

L'exécution est par défaut synchrone, mais on peut configurer un modèle asynchrone



29



30



Configuration asynchrone

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

Cette interface définit une méthode :

```
void execute(Runnable task)
```

Ex :

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



Paramètres de Job

Les paramètres d'un Job sont fournis via ***JobParameters*** qui est une *Map<String, JobParameter>*

Il est important qu'un *JobParameters* puisse être comparé de manière fiable à un autre pour l'égalité, afin de déterminer si le job est dans les mêmes conditions de démarrage.



JobParameter

JobParameter représente un paramètre d'un job.

- Seuls les types suivants peuvent être des paramètres: ***String***, ***Long***, ***Date*** et ***Double***.
- Le flag ***identifying*** indique si le paramètre doit être utilisé dans le cadre de l'identification d'une instance de job.



Validateur de paramètres

Un job peut déclarer un bean implémentant

JobParametersValidator pour valider les paramètres

- Une implémentation est disponible :
DefaultJobParametersValidator
Il peut combiner les contraintes de paramètres obligatoires et facultatifs simples.



CommandLineJobRunner

SpringBatch 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
public class SpringBootBatchProcessingApplication implements
    CommandLineRunner {

    public static void main(String[] args) {
        SpringApplication.run(SpringBootBatchProcessingApplication.class, args);
    }
    @Override
    public void run(String... args) throws Exception {
        // Récupération des arguments et démarrage du job via jobLauncher
    }
}

...
./mvnw clean package
...
java -jar myBatch.jar <myArgs>
```




ExitCode

Le traitement batch g r  par des schedulers n cessite de retourner des codes num riques :

- En g n ral 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 propri t  de type String est converti par un bean ***ExitCodeMapper***
- L'impl mentation par d faut *SimpleJvmExitCodeMapper* retourne ;
 - 0 pour succ s
 - 1 pour les erreurs g n riques
 - 2 pour les erreurs de job runner comme par exemple (Impossible de trouver le job)
- Il est possible d'impl menter son propre *ExitCodeMapper*

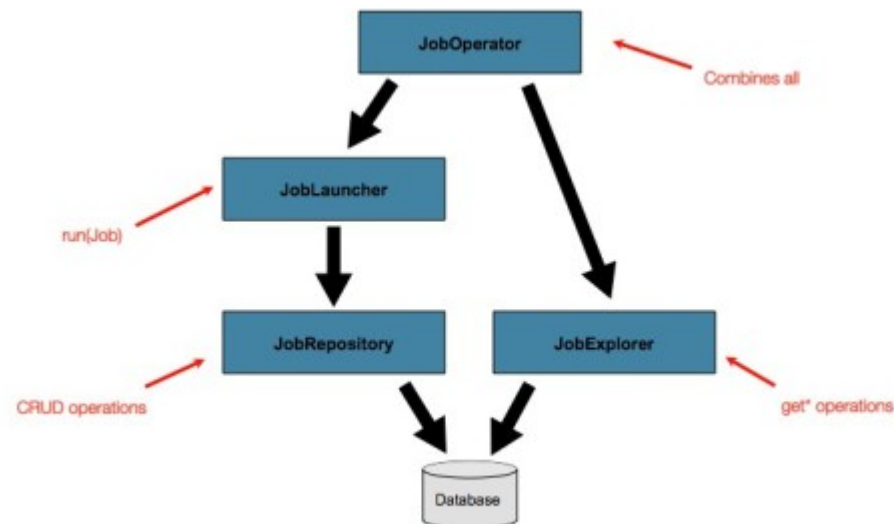


Jobs

Démarrage
Accès aux méta-données

Introduction

Lorsque l'on doit gérer de nombreux jobs et des contraintes de scheduling plus complexes, les interfaces ***JobOperator*** et ***JobExplorer*** permettent de surveiller et contrôler les exécution des jobs





Interface *JobExplorer*

JobExplorer fournit les méthodes permettant d'inspecter le contenu de la base et visualiser les exécutions stockées.

```
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);  
}
```



JobRegistry

Un ***JobRegistry*** est une interface qui permet de recenser tous les jobs enregistrés dans l'application

L'implémentation par défaut fournie par SB est :
MapJobRegistry

Afin que les jobs de l'application soit automatiquement enregistré dans JobRegistry. Il faut cependant fournir un bean ***JobRegistryBeanPostProcessor***

```
@Bean
public JobRegistryBeanPostProcessor jobRegistryBeanPostProcessor() {
    JobRegistryBeanPostProcessor postProcessor =
        new JobRegistryBeanPostProcessor();
    postProcessor.setJobRegistry(jobRegistry());

    return postProcessor;
}
```



JobOperator

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

Il s'appuie sur un *JobRegistry* ayant enregistré les Jobs disponibles

Il dépend également de :

- *JobRepository*
- *JobExplorer*
- *JobLauncher*

Une implémentation courante à fournir est
SimpleJobOperator



Configuration XML

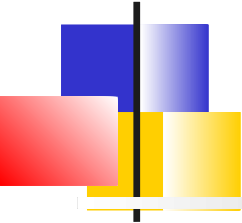
```
<bean id="jobOperator" class="org.spr...SimpleJobOperator">
  <property name="jobExplorer">
    <bean class="org.spr...JobExplorerFactoryBean">
      <property name="dataSource" ref="dataSource" />
    </bean>
  </property>
  <property name="jobRepository" ref="jobRepository" />
  <property name="jobRegistry"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

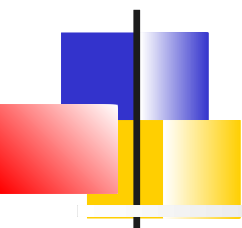
Interface *JobOperator*

```
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());
```

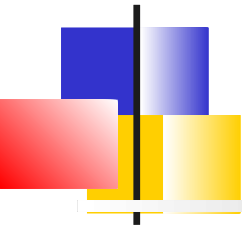


JobParametersIncrementer

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

Traitement par morceau

Redémarrage / Skip / Retry

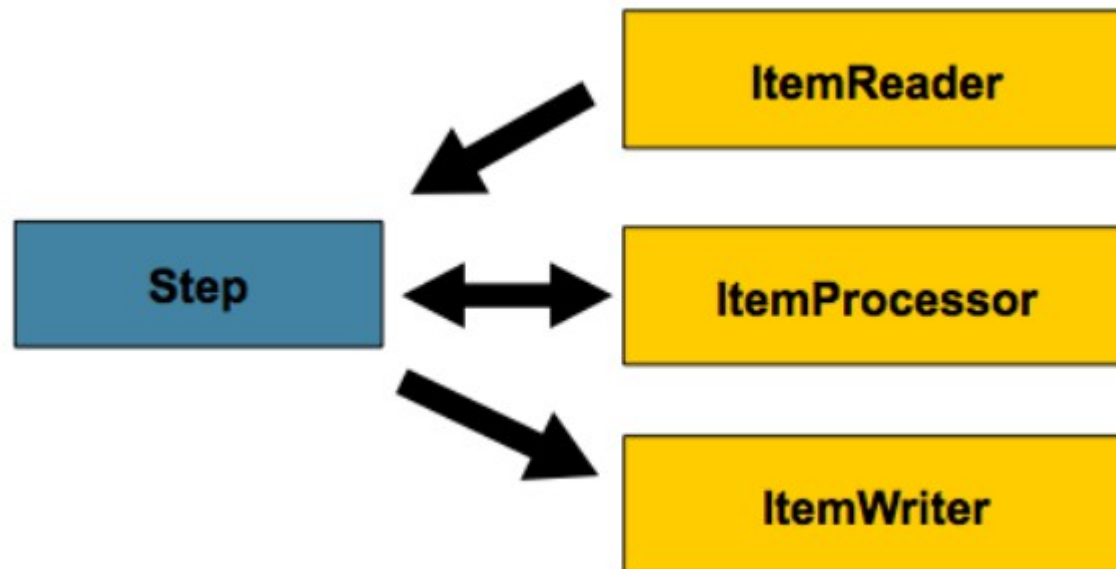
Scopes

Listeners

StepFlow



Composants d'un *Step*



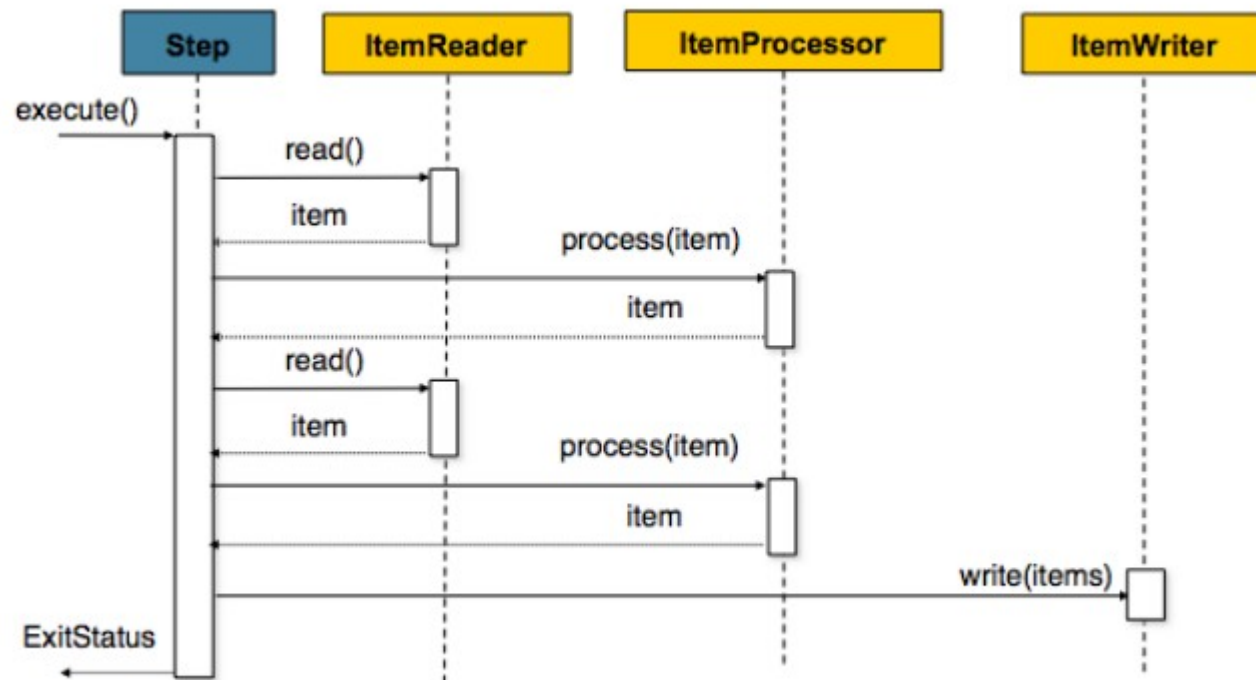


Chunk-oriented

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

Ratio read/write





Logique d'un step

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




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>
```

Java

@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 validée.



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

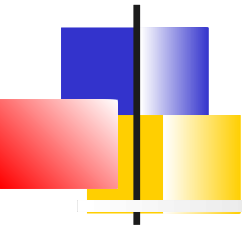
```
<!-- Configuration incomplète (pas de reader, pas de writer)  
Alors la step est abstraite -->
```

```
<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>
```



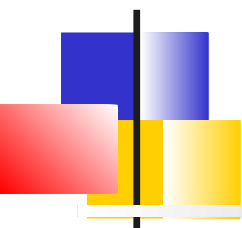
Fusion de liste

```
<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>
```



Configuration des steps

Traitement par morceau
Redémarrage / Skip / Retry
Scopes disponibles
Listeners
StepFlow



Contraintes de redémarrage

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)`
- Pouvoir redémarrer 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 démarrée 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 élément en cas d'erreur.

If faut configurer les exceptions et leurs nombres 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

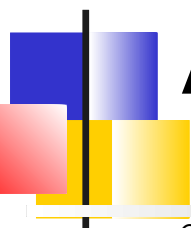
```
@Bean
public Step step1() {
    return this.stepBuilderFactory.get("step1")
        .<String, String>chunk(2)
        .reader(itemReader())
        .writer(itemWriter())
        .faultTolerant()
        .noRollback(ValidationException.class)
        .build();
}
```



Attributs de transaction

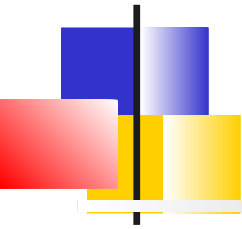
Les attributs de transaction peuvent être utilisés pour contrôler les paramètres d'isolation, de propagation et de timeout.

```
<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();
}
```



Configuration des steps

Traitement par morceau
Redémarrage / Skip / Retry

Scopes

Listeners

StepFlow



Introduction

Les données requises par les composants des steps ne sont pas toujours connues au moment de la compilation.

Il est possible par exemple de récupérer une propriété système positionnée au moment de l'exécution via - *D* :

```
@Bean
public FlatFileItemReader flatFileItemReader(@Value("${input.file.name}")
String name) {
    return new FlatFileItemReaderBuilder<Foo>()
        .name("flatFileItemReader")
        .resource(new FileSystemResource(name))
        ...
}
```




Paramètres et scopes

Cependant, il est préférable d'utiliser les paramètres de Job pour passer des données dynamiques

SpringBatch permet de définir 2 scopes Spring influant sur le cycle de vie du bean :

- **StepScope** : Le bean est créé au démarrage de la step
- **JobScope** : Le bean est créé au démarrage du Job

Ces 2 *scopes* peuvent permettre d'initialiser les propriétés du bean au moment de sa création

- => Les composants peuvent alors accéder aux *JobParameters* ou au contexte d'exécution



Exemple *@StepScope*

@StepScope

@Bean

public FlatFileItemReader

flatFileItemReader(@Value("#{jobParameters['input.file.name']}") String name) {

...

}

@StepScope

@Bean

public FlatFileItemReader

flatFileItemReader(@Value("#{jobExecutionContext['input.file.name']}") String name) {

...

}

@StepScope

@Bean

public FlatFileItemReader

flatFileItemReader(@Value("#{stepExecutionContext['input.file.name']}") String name)

{

...

}



Exemples *@JobScope*

@JobScope

@Bean

```
public FlatFileItemReader flatFileItemReader(  
    @Value("#{jobParameters[input]}") String name) {
```

```
    return new FlatFileItemReaderBuilder<Foo>()  
        .name("flatFileItemReader")  
        .resource(new FileSystemResource(name))  
        ...
```

```
}
```

@JobScope

@Bean

```
public FlatFileItemReader flatFileItemReader(  
    @Value("#{jobExecutionContext['input.name']}") String name) {
```

```
    return new FlatFileItemReaderBuilder<Foo>()  
        .name("flatFileItemReader")  
        .resource(new FileSystemResource(name))  
        ...
```

```
}
```



Configuration XML

Les scopes *StepScope* et *JobScope* sont automatiquement disponibles si on utilise *@EnableBatchProcessing* ou Spring Boot

Lors d'une configuration XML, il faut explicitement les déclarer en tant que bean

```
<bean class="org.springframework.batch.core.scope.JobScope" />  
<bean class="org.springframework.batch.core.scope.StepScope" />
```



Passer des données entre les steps

Lors de l'exécution d'un step, les données à sauvegarder après chaque commit sont stockées dans le *StepExecution*¹

Si certaines données doivent être passées à un step ultérieur, elles doivent être promues dans le *JobExecution* à la fin du step

ExecutionContextPromotionListener

permet de définir les clés du *StepExecution* qui doivent être promues dans le *JobExecution*

1. Ne pas utiliser le *JobExecution* car les données seraient perdues si le step échoue



Configuration

```
@Bean
public Step step1() {

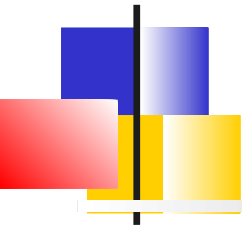
    return this.stepBuilderFactory.get("step1")
        .<String, String>chunk(10)
        .reader(reader())
        .writer(savingWriter())
        .listener(promotionListener())
        .build();
}

@Bean
public ExecutionContextPromotionListener promotionListener() {
    ExecutionContextPromotionListener listener = new
    ExecutionContextPromotionListener();
    listener.setKeys(new String[] {"someKey"});
    return listener;
}
```



Récupérer les données

```
public class RetrievingItemWriter implements ItemWriter<Object> {  
    private Object someObject;  
    public void write(List<? extends Object> items) throws Exception {  
        // ...  
    }  
  
    @BeforeStep // Interface StepListener  
    public void retrieveInterstepData(StepExecution stepExecution) {  
        JobExecution jobExecution = stepExecution.getJobExecution();  
        ExecutionContext jobContext = jobExecution.getExecutionContext();  
        this.someObject = jobContext.get("someKey");  
        stepExecution.getExecutionContext().put("someKey", this.someObject) ;  
    }  
}
```



Configuration des steps

Traitement par morceau
Redémarrage / Skip / Retry
Scopes
Listeners
StepFlow



Listeners

Il est possible d'associer des ***listeners*** des événements liés aux steps

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

Les listeners :

- Soit implémentent une interface étendant *StepListener*
- Soit contiennent des méthodes annotées avec les annotations de *StepListener*

Les listeners permettent de manipuler le contexte d'Exécution (job ou step)



Configuration

XML

```
<step id="step1">
  <tasklet>
    <chunk reader="reader" writer="writer" commit-interval="10"/>
    <listeners>
      <listener ref="chunkListener"/>
    </listeners>
  </tasklet>
</step>
```

Java

```
@Bean
public Step step1() {

    return this.stepBuilderFactory.get("step1")
        .<String, String>chunk(10)
        .reader(reader())
        .writer(writer())
        .listener(chunkListener())
        .build();
}
```



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

ItemWriterListener 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.

Les méthodes sont appelées au moment de la validation

```
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

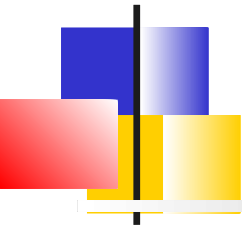
```
@Bean
public Step step1() {
    return this.stepBuilderFactory.get("step1")
        .tasklet(myTasklet())
        .build();
}
```



TaskletAdpater

TaskletAdapter permet d'utiliser une classe existante

```
@Bean
public MethodInvokingTaskletAdapter myTasklet() {
    MethodInvokingTaskletAdapter adapter = new
        MethodInvokingTaskletAdapter();
    adapter.setTargetObject(fooDao());
    adapter.setTargetMethod("updateFoo");
    return adapter;
}
```



Configuration des steps

Traitement par morceau
Redémarrage / Skip / Retry
Scopes
Listeners
StepFlow



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 le résultat 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();
}
```



Échouer 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 redémarré.

```
<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">
```

```
return new JobBuilder("job", jobRepository)
.start(step1)
.next(step2).on("FAILED").fail()
.from(step2).on("*").to(step3)
.end()
.build();
```



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 complexes 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();
}
```



Exécution parallèle

SpringBatch permet de configurer un job avec des exécutions parallèles via l'opérateur ***split***.

```
<!-- 2 branches // arrivent en step4. B1 = step1,step2. B2=step3 -->
<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 splitFlow() {
    return new FlowBuilder<SimpleFlow>("splitFlow")
        .split(new SimpleAsync)
        .add(flow1(), flow2())
        .build();
}

@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(splitFlow)
        .next(step4())
        .end().build();
}
```



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***¹ 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();  
}
```



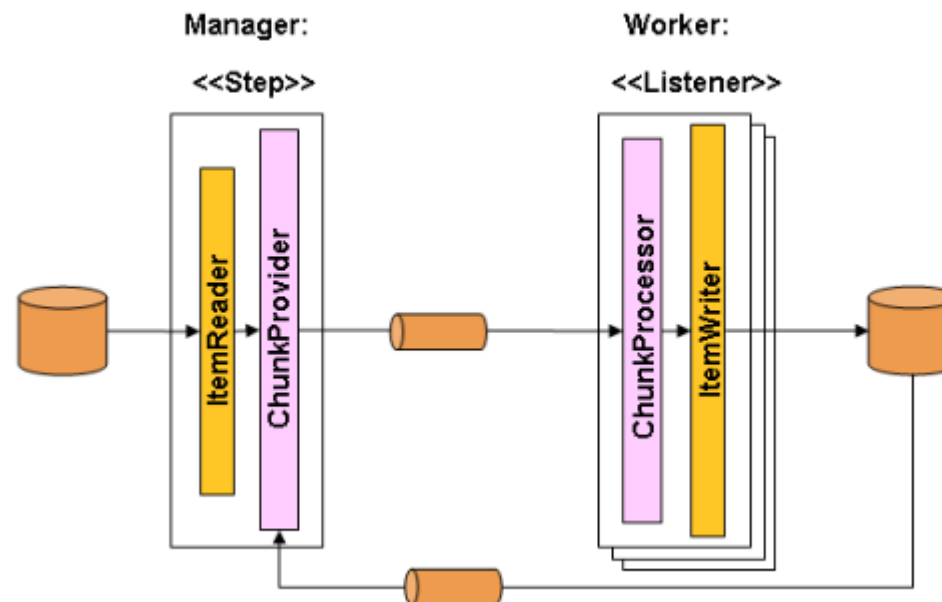
Steps en parallèle

```
@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

Un processus contient l'ItemReader et envoie des Chunk vers un middleware (Typiquement un message Broker)

Le broker distribue les chunks vers plusieurs consommateurs qui inclut le *ItemProcessor* et le *ItemWriter*

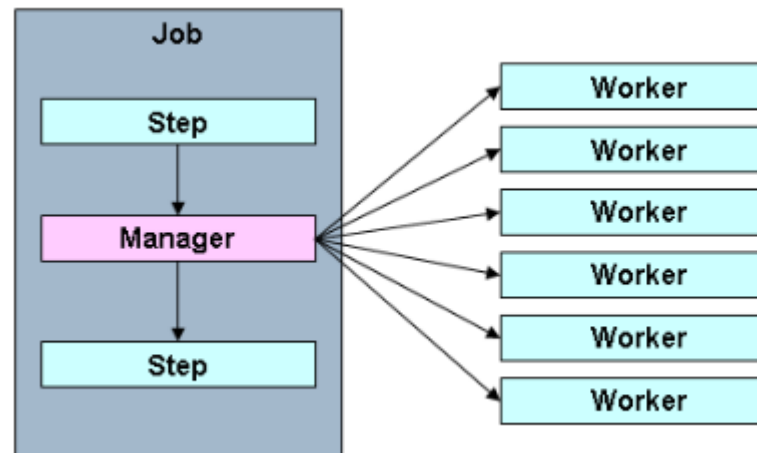


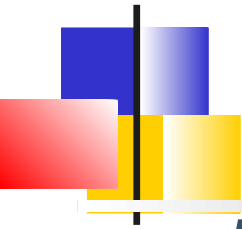
Partitionnement

Un Step est un ***Manager***

Les *workers* (locaux ou distants) exécutent la step pour 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 à l'intérieur 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 *doInIteration*, 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

Implémentations de RetryPolicy :

- ***SimpleRetryPolicy*** permet une nouvelle tentative en fonction
 - d'une liste d'Exception acceptées avec pour chacune un nombre de 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

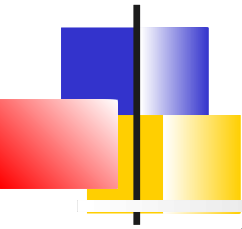
Scaling et traitement parallèle

Répétition

Ré-essai

Tests unitaires

Patterns classiques



Versions

Spring/SpringBoot/JUnit

SpringBoot 1, Spring 4, JUnit4

Dernière version Septembre 2018

SpringBoot 2, Spring 5, JUnit5

Première version ~2018



Rappels *spring-test*

Spring Test apporte peu pour le test unitaire

- Mocking de l'environnement en particulier l'API servlet ou Reactive
- Package d'utilitaires : `org.springframework.test.util`

Et beaucoup pour les tests d'intégration (impliquant un `ApplicationContext` Spring) :

- Cache du conteneur Spring pour accélérer les tests
- Injection des données de test
- Gestion de la transaction (roll-back)
- Des classes utilitaires
- Intégration JUnit4 et JUnit5



Intégration JUnit

Pour JUnit4 :

@RunWith(SpringJUnit4ClassRunner.class)

ou ***@RunWith(SpringRunner.class)***

Permet de charger un contexte Spring, effectuer l'injection de dépendances, etc.

Pour JUnit5 :

@ExtendWith(SpringExtension.class)

Permet aussi de charger un contexte Spring, effectuer l'injection de dépendances, etc.

Et en plus de l'injection de dépendance pour les méthodes de test, des conditions d'exécution en fonction de la configuration Spring, des annotations supplémentaires pour gérer les transactions



Exemple JUnit5

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = TestConfig.class)
class SimpleTests {

    @Test
    void testMethod() {
        // test logic...
    }
}
```



@SpringBatchTest

Grâce à l'annotation

@SpringBatchTest, des utilitaires de test pour le batch sont disponible dans le contexte de test :

- ***JobLauncherTestUtils*** (nécessite un bean Job) : Test de bout en bout des steps individuellement
- ***JobRepositoryTestUtils*** : Permet de créer puis supprimer des instances de *JobExecution* d'une base de données



Example - JUnit4

```
@SpringBatchTest
@RunWith(SpringRunner.class)
@ContextConfiguration(classes=SkipSampleConfiguration.class)
public class SkipSampleFunctionalTests { ... }
```

```
@SpringBatchTest
@RunWith(SpringRunner.class)
@ContextConfiguration(
    locations = { "/simple-job-launcher-context.xml",
                  "/jobs/skipSampleJob.xml" })
public class SkipSampleFunctionalTests { ... }
```



Test complet du batch

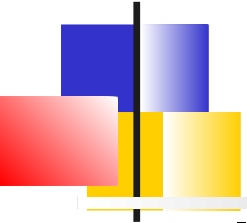
Un test complet consiste principalement :

- Initialiser la source avec des données de test
- Démarrer le job via la méthode *launchJob(JobParameters)* de *JobLauncherTestUtils*
- Récupérer le *JobExecution* retourné et y faire des assertions



Example

```
@SpringBatchTest
@RunWith(SpringRunner.class)
@ContextConfiguration(classes=SkipSampleConfiguration.class)
public class SkipSampleFunctionalTests {
    @Autowired
    private JobLauncherTestUtils jobLauncherTestUtils;
    private SimpleJdbcTemplate simpleJdbcTemplate;
    @Autowired
    public void setDataSource(DataSource dataSource) { this.simpleJdbcTemplate = new
        SimpleJdbcTemplate(dataSource);
    }
    @Test
    public void testJob() throws Exception {
        simpleJdbcTemplate.update("delete from CUSTOMER");
        for (int i = 1; i <= 10; i++) {
            simpleJdbcTemplate.update("insert into CUSTOMER values (?, 0, ?, 100000)", i, "customer"
                + i);
        }
        JobExecution jobExecution = jobLauncherTestUtils.launchJob();
        Assert.assertEquals("COMPLETED", jobExecution.getExitStatus().getExitCode());
    }
}
```



Tester individuellement les étapes

Il est souvent plus simple de tester individuellement les étapes.

L'utilitaire *jobLauncherTestUtils* permet de lancer une seule step via sa méthode ***launchStep()***

```
JobExecution jobExecution =  
jobLauncherTestUtils.launchStep("loadFileStep");
```

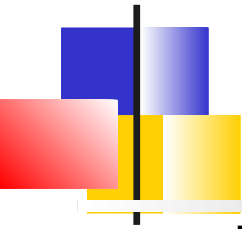


JobScope/StepScope

Les composants des étapes utilisent généralement le StepScope pour résoudre leur propriétés

SpringBatch fournit 2 composants qui permettent de contrôler le contexte d'exécution Job ou Step lors d'un test :

- ***StepScopeTestExecutionListener / JobScopeTestExecutionListener*** :
Responsable de créer un *StepExecution/JobExecution* pour chaque méthode de test
- ***StepScopeTestUtils/JobScopeTestUtils*** :
Utilitaire pour manipuler le *StepScope/JobScope*



Step/Job- TestExecutionListener

- Lors de l'exécution du test, les
**TestExecutionListener* recherchent
une méthode dans le cas de test qui
renvoie un *StepExecution/JobExecution*
- Si il la trouve, il l'utilise pour le test
 - Si aucune méthode n'existe, un
StepExecution/JobExecution par défaut
est créée.



Exemple

```
@SpringBatchTest
@RunWith(SpringRunner.class)
@ContextConfiguration
public class StepScopeTestExecutionListenerIntegrationTests {
    // ItemReader est défini en step-scoped,
    // il ne peut être injecté que lorsqu'un step est actif
    @Autowired
    private ItemReader<String> reader;
    public StepExecution getStepExecution() {
        StepExecution execution = MetaDataInstanceFactory.createStepExecution();
        execution.getExecutionContext().putString("input.data", "foo,bar,spam");
        return execution;
    }
    @Test
    public void testReader() {
        // Le reader est initialisé et associé à ses données d'entrée
        assertNotNull(reader.read());
    }
}
```



StepScopeTestUtil

StepScopeTestUtil permet de tester les Reader/Writer dépendant d'un StepScope en créant un scope de test via MetadataInstanceFactory

L'utilitaire propose une méthode *doInTestScope()* prenant 2 paramètres :

- *stepExecution*
- Une lambda Callable qui manipule le Reader ou Writer à tester



Example

```
@Test
public void givenMockedStep_whenReaderCalled_thenSuccess() throws Exception {
    // given
    StepExecution stepExecution =
        MetadataInstanceFactory.createStepExecution(testJobParameters());
    // when
    StepScopeTestUtils.doInStepScope(stepExecution, () -> {
        BookRecord bookRecord;
        itemReader.open(stepExecution.getExecutionContext());
        while ((bookRecord = itemReader.read()) != null) {
            // then
            assertThat(bookRecord.getBookName(), is("Foundation"));
            assertThat(bookRecord.getBookAuthor(), is("Asimov I.));
        }
        itemReader.close();
        return null;
    });
}
```



Vérification des fichiers de sortie

Spring Batch fournit ***AssertFile*** pour faciliter la vérification des fichiers de sortie

```
private static final String EXPECTED_FILE =  
    "src/main/resources/data/input.txt";  
private static final String OUTPUT_FILE =  
    "target/test-outputs/output.txt";
```

```
AssertFile.assertFileEquals(  
    new FileSystemResource(EXPECTED_FILE),  
    new FileSystemResource(OUTPUT_FILE));
```



Mock

Quelquefois un test nécessite des dépendances de beans qui ne sont pas nécessaires pour la logique de test

MetaDataInstanceFactory fournit des méthodes pour créer des instances de test pour *JobExecution*, *JobInstance*, *StepExecution*.



Pour aller plus loin

Scaling et traitement parallèle

Répétition

Ré-essai

Tests unitaires

Patterns classiques



Tracer les erreurs

```
public class ItemFailureLoggerListener extends ItemListenerSupport {  
  
    private static Log logger = LogFactory.getLog("item.error");  
    public void onReadError(Exception ex) {  
        logger.error("Encountered error on read", e);  
    }  
    public void onWriteError(Exception ex, List<? extends Object> items) {  
        logger.error("Encountered error on write", ex);  
    }  
}
```




Arrêter un job

Il se peut que les données d'entrée nécessite d'arrêter un job en cours.

Plusieurs possibilités :

- Envoyer une RuntimeException qui n'est pas réessayable
- Renvoyer null dans le Reader et garder la CompletionPolicy par défaut
- Implémenter un listener qui appelle `stepExecution.setTerminateOnly();`



Ajouter un footer dans le fichier de sortie

FlatFileFooterCallback (et *FlatFileHeaderCallback*) sont des propriétés optionnelles de *FlatFileItemWriter*

En général, on écrit des données agrégées dans le footer



Exemple

```
public class TradeItemWriter implements ItemWriter<Trade>,
FlatFileFooterCallback {
    private ItemWriter<Trade> delegate;
    private BigDecimal totalAmount = BigDecimal.ZERO;
    public void write(List<? extends Trade> items) throws Exception {
        BigDecimal chunkTotal = BigDecimal.ZERO;
        for (Trade trade : items) {
            chunkTotal = chunkTotal.add(trade.getAmount());
        }
        delegate.write(items);
        // Après le commit
        totalAmount = totalAmount.add(chunkTotal);
    }
    public void writeFooter(Writer writer) throws IOException {
        writer.write("Total Amount Processed: " + totalAmount);
    }
}
```

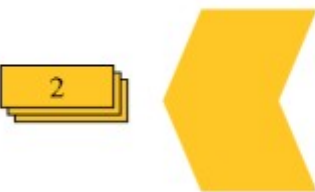


Driving Query

L'approche « **Driving Query** » pour les BD consiste à itérer sur les clés plutôt que sur les objets entier.

- Cela permet de régler des problèmes du à un curseur trop grand pour certains types de BD

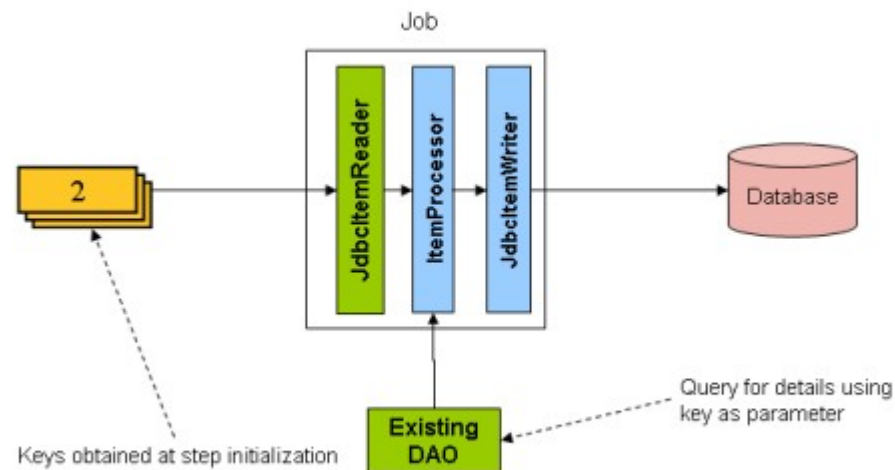
Select ID from F00
where id > 1 and id < 7

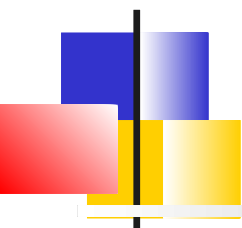


ID	NAME	BAR
1	foo1	bar1
2	foo2	bar2
3	foo3	bar3
4	foo4	bar4
5	foo5	bar5
6	foo6	bar6
7	foo7	bar7
8	foo8	bar8

Driving Query

Avec cette approche, l'objet est complet est retourné à partir de sa clé dans un *ItemProcessor*





Enregistrement multi-lignes

Pour traiter les enregistrements multi-lignes, il faut implémenter un *ItemReader* qui encapsule un *FlatFileItemReader*

@Bean

```
public MultiLineTradeItemReader itemReader() {  
    MultiLineTradeItemReader itemReader = new  
        MultiLineTradeItemReader();  
    itemReader.setDelegate(flatFileItemReader());  
    return itemReader;  
}
```



Exemple

```
private FlatFileItemReader<FieldSet> delegate;
public Trade read() throws Exception {
    Trade t = null;
    for (FieldSet line = null; (line = this.delegate.read()) != null;) {
        String prefix = line.readString(0);
        if (prefix.equals("HEA")) { // Ligne d'entête de l'enregistrement
            t = new Trade();
        } else if (prefix.equals("NCU")) { // Ligne intermédiaire de données
            Assert.notNull(t, "No header was found.");
            t.setLast(line.readString(1));
            ...
        } else if (prefix.equals("FOT")) { // Footer marquant la fin d'enregistrement
            return t;
        }
    }
    Assert.isNotNull(t, "No 'END' was found.");
    return null;
}
```



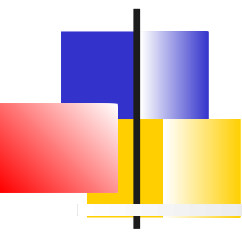
Commandes système

Spring Batch fournit

SystemCommandTasklet : une implémentation de *Tasklet* pour appeler les commandes systèmes.

@Bean

```
public SystemCommandTasklet tasklet() {  
    SystemCommandTasklet tasklet = new SystemCommandTasklet();  
    tasklet.setCommand("echo hello");  
    tasklet.setTimeout(5000);  
    return tasklet;  
}
```

Merci pour votre attention !!

Références SpringBatch Reference

<https://docs.spring.io/spring-batch/docs/current/reference/html/index.html>



Introduction

Batch Processing

Spring Batch : Architecture et concepts

Rappels SpringBoot

Spring Batch et *Spring Boot*



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

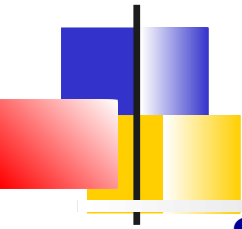


Auto-configuration

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

SpringBoot est capable de détecter automatiquement la nature de l'application¹ 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



Spring Initializr

Spring Initializr est un assistant pour la création de projet SpringBoot

- Disponible via une API web, il est intégré dans les IDEs ainsi que dans une application web (<https://start.spring.io/>)

Il permet :

- De sélectionner la version de SpringBoot qui fixera toutes les autres version des librairies du projet
- La version de Java, L'outil de build Maven ou Gradle
- Des starter-modules : Groupe de dépendances pour une fonctionnalité applicative

Il génère ensuite :

- Un fichier de dépendance Maven ou Gradle
- Une arborescence projet avec une classe principale permettant de démarrer et une classe de test



Structure projet

Aucune obligation mais des recommandations :

- Placer la classe Main dans le package racine
- L'annoter avec **@SpringBootApplication** qui englobe :
 - **@EnableAutoConfiguration** : Activation de SpringBoot
 - **@ComponentScan** : Point de départ pour le scan de packages afin de trouver les annotations Spring
 - **@Configuration** : Classe de configuration pouvant contenir des méthodes de création de bean



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 principale est exécutable, ce qui veut dire que l'application peut être démarrée en tant qu'application Java

Les plugins Maven et Gradle de Boot permettent de produire un "fat jar" exécutable

```
mvn package, gradle build
```

```
java -jar <path-to-jar>
```




Annotations pour la définition de beans et l'injection de dépendance

Définition de beans :

- Annoter une classe présente dans un sous-package de *@ComponentScan* avec **@Component**, **@Service**, **@Repository**, **@Controller**
- Annoter une méthode avec **@Bean** dans une classe annotée par *@Configuration*

Injection de dépendance

- Injection implicite via le constructeur
- L'annotation **@Autowired** permet d'injecter une dépendance via son type
- L'annotation **@Resource** permet d'injecter une dépendance via son nom



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 implicate
    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 externes** (.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
De remplacer les beans auto-configurés par sa propre implémentation.
- La **désactivation** de l'auto-configuration
- ...