

SPRING BATCH

Présentation



PRÉSENTATION

- + Framework qui respecte la norme batch du Java 7
 - Mais fonctionne depuis Java 4
- + Objectifs : Exécuter des batch (au sens production)
 - Remplace les scripts jcl, ksh, bat, sh, ...
- + Ne remplace pas un ordonnanceur (\$Univers, cron, ...)
- + Le framework batch du Spring est à part
 - Il n'est pas avec le Spring Framework
 - Il faut l'ajouter dans ses dépendances (maven ou autre)

PRÉSENTATION

- + Actuellement Spring Batch est dans sa version 3.0.7
- + Il se compose de deux librairies (JAR)
 - batch-core : le cœur de l'API
 - batch-infrastructure : les éléments techniques de l'API
- + Pour fonctionner Spring Batch aura besoin du Spring framework (plus ou moins de modules selon les batch)
 - **ATTENTION** : Toutes les versions de spring-batch ne sont pas compatibles avec toutes les versions de Spring framework
 - Selon les traitements de vos batch et la version de spring-batch, vous aurez aussi besoin d'autres modules du Spring (par exemple spring-xml)
 - Potentiellement, d'autres JAR seront aussi indispensables (commons, drivers, ...)

RAPPEL UN BATCH

- + Un batch est un ensemble de **job** = tâches
- + Un job se composera de **step** = étapes
- + Lors de son exécution un job doit réaliser toutes les *step*
- + Si tout va bien, le job se termine sur sa dernière *step* avec un **status COMPLETED**
- + Si un problème survient
 - Le *job* peut être relancé juste à partir de la *step* qui a posé un problème
 - Le *job* peut être relancé dans sa totalité

RAPPEL UN BATCH

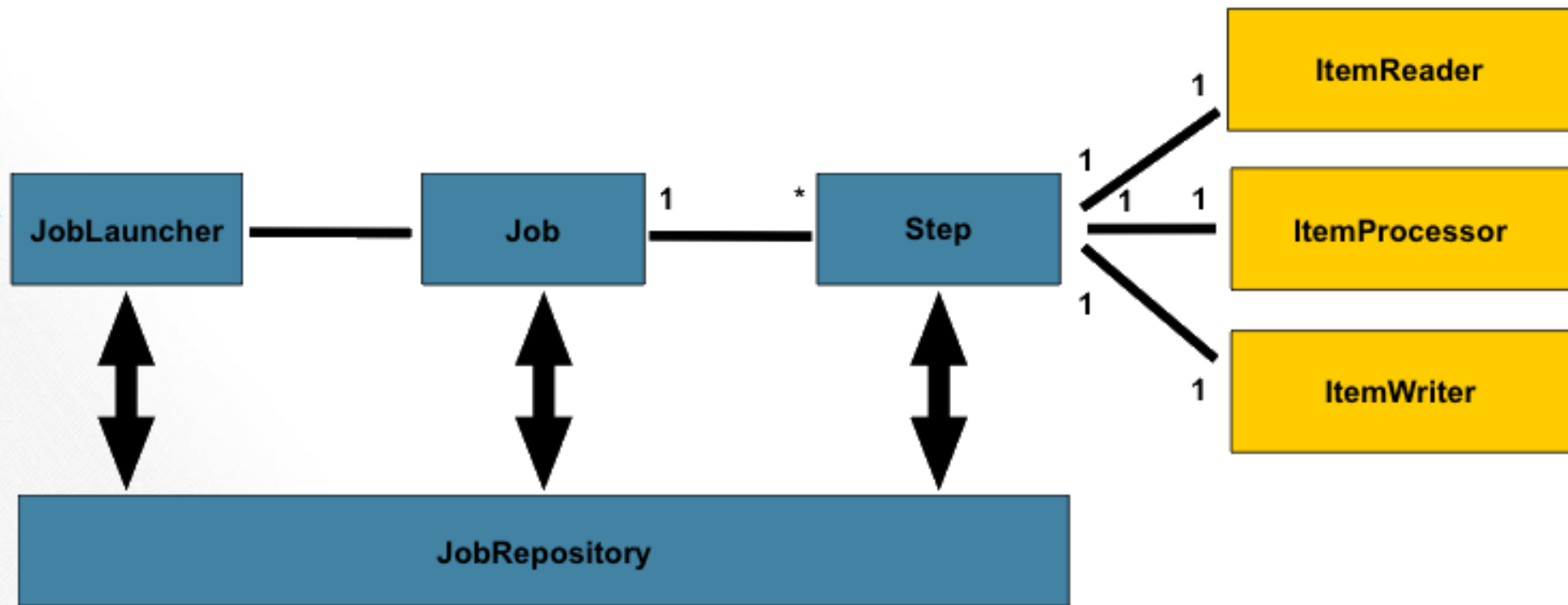
- + Lors du lancement d'un batch vous pouvez ajouter des **paramètres**
 - En général on indique au moins la date du lancement
- + Un batch se lance généralement via un ordonnanceur
 - Cron
 - \$Univers
 - Services Windows
 - ...
- + C'est l'ordonnanceur
 - qui décide la période et les horaires de lancement du batch
 - qui fait passer les paramètres au batch
 - qui récupère les informations associés aux déroulements du batch

EXEMPLE D'UN TRAITEMENT BATCH

1. Récupération d'information dans une base de données
 - Ou dans un ou plusieurs fichiers
2. Traitement des données
 - Trier, supprimer, réassembler ...
3. Réalisation d'un ou plusieurs fichiers résultats
 - Ou mise à jour d'une base de données

EN SPRING BATCH

+ Fonctionnement d'un Batch



EN SPRING BATCH

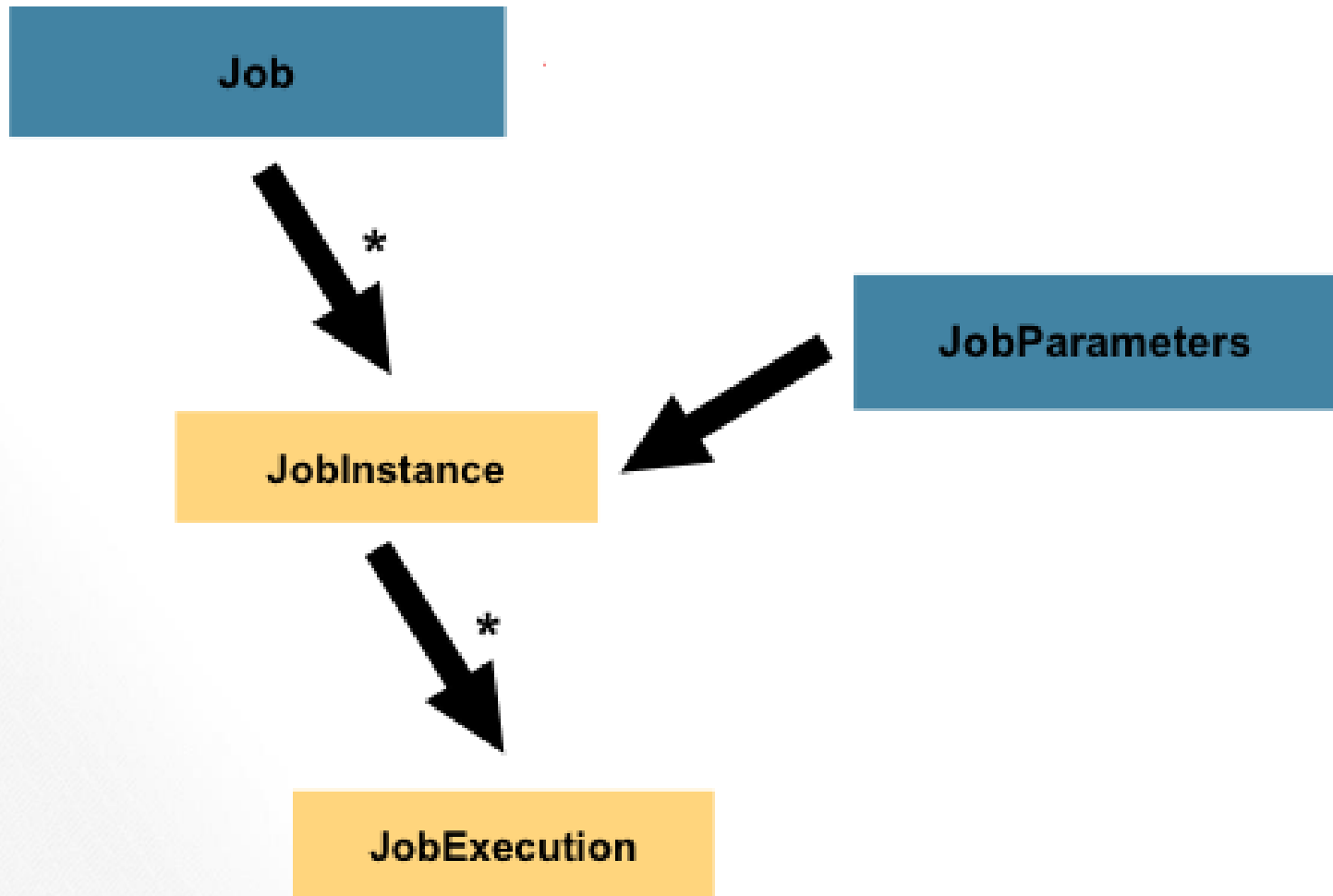
- + Les *Job* seront automatiquement associés à un *JobRepository*
- + Un *JobRepository* a la responsabilité de gérer les informations associés aux *Job* (paramètres, états, transactions, ...)
- + Un *JobRepository* peut stoker ses informations en mémoire ou dans une base de données
- + Un *JobRepository* sera associé manuellement à un *JobLauncher*
 - Il a la responsabilité de gérer les états des *Job*

EN SPRING BATCH

- + Un *Job* sera composé de 1..n *Step*
- + Un *Step* sera composé d'une *tasklet* qui aura
 - Un seul **ItemReader** : entrée des données
 - 0 ou Un **ItemProcessor** : traitement des données
 - Un seul **ItemWriter** : sortie des données

EN SPRING BATCH

+ Fonctionnement logique et technique



EN SPRING BATCH

- + **JobParameters** : Ensemble de paramètres utilisé lors du lancement du *Job*
- + **JobInstance** : concept *logique* associé au lancement d'un *Job*
 - Représente l'association : *Job* + valeurs *JobParameters*
 - Peut générer plusieurs *JobExecution*
- + **JobExecution** : concept technique associé au lancement d'un *Job*.
 - Pour un *JobInstance* on peut avoir plusieurs *JobExecution*, l'un se terminant correctement et pas l'autre par exemple

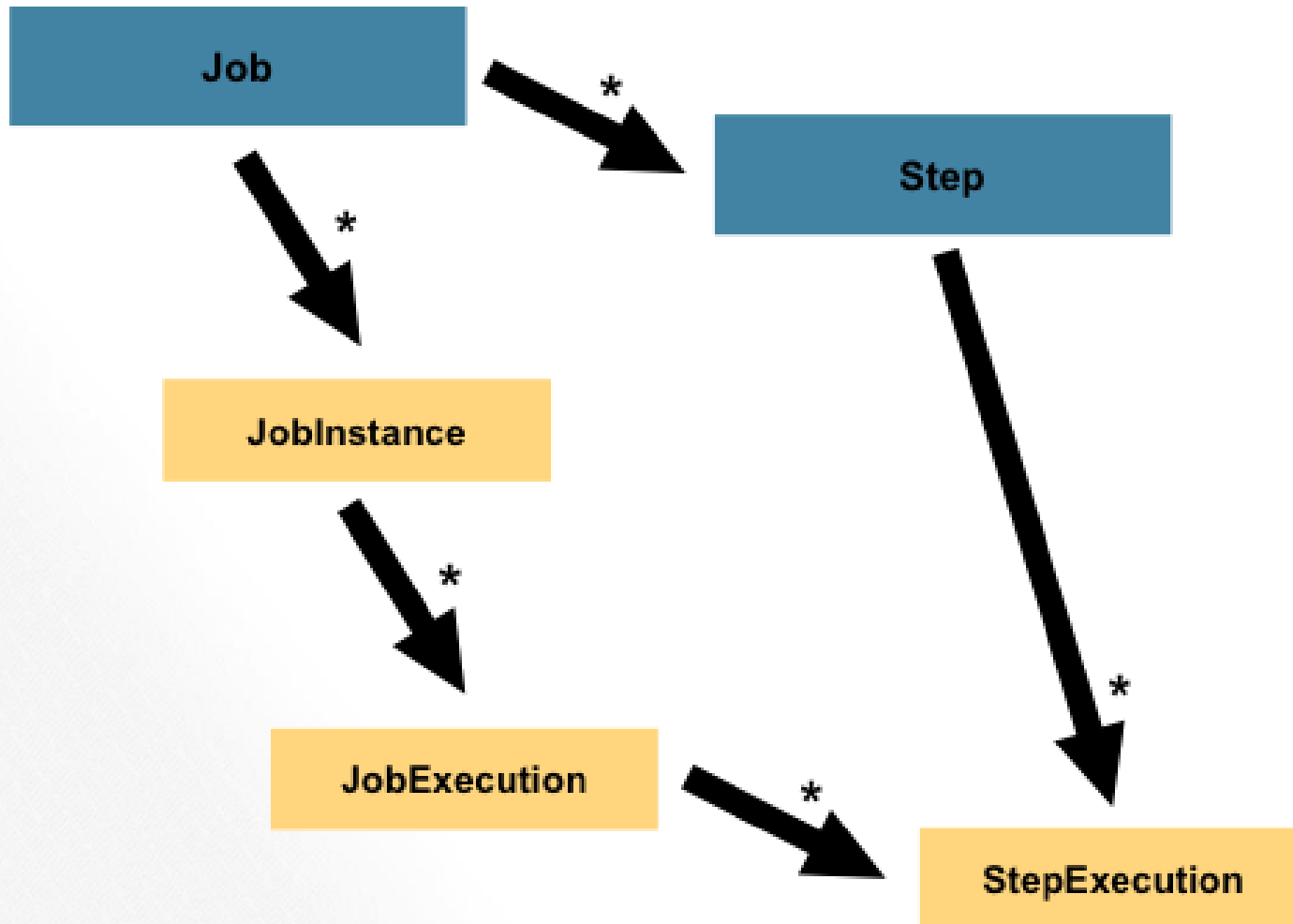
JOBEXECUTION

+ Chaque *JobExecution* contiendra les informations suivantes :

- **status** : une valeur associée à BatchStatus qui représente le status à l'instant demandé (ex: BatchStatus.STARTED, BatchStatus.FAILED, BatchStatus.COMPLETED)
- **startTime** : java.util.Date de départ
- **endTime** : java.util.Date de fin, peut importer la réussite ou l'échec du Job.
- **exitStatus** : status à la fin du Job (ne pas la confondre avec status)
- **createTime** : java.util.Date représentant la création du JobExecution. N'a pas forcément de rapport avec startTime.
- **lastUpdated** : java.util.Date représentant la mise à jour du JobExecution.
- **executionContext** : espace permettant de stocker des informations diverses
- **failureExceptions** : Liste d'erreurs associée à l'exécution du Job.

STEP ET STEPEXECUTION

+ De la même manière, concernant les *Step*



STEP ET STEPEXECUTION

- + Le *Job* est composé de *Step*
- + **StepExecution** : concept technique associé au lancement d'un *Step*.
 - Pour un chaque exécution d'un *Step* on aura un *StepExecution*

STEPEXECUTION

- + Chaque *StepExecution* contiendra les informations suivantes :
 - **status** : une valeur associé à BatchStatus qui représente le status à l'instant demandé (ex: BatchStatus.STARTED, BatchStatus.FAILED, BatchStatus.COMPLETED)
 - **startTime** : java.util.Date de départ
 - **endTime** : java.util.Date de fin, peut importe le status.
 - **exitStatus** : status à la fin du Step (ne pas la confondre avec status)
 - **executionContext** : espace permettant de stoker des informations diverses

STEPEXECUTION

+ Suite

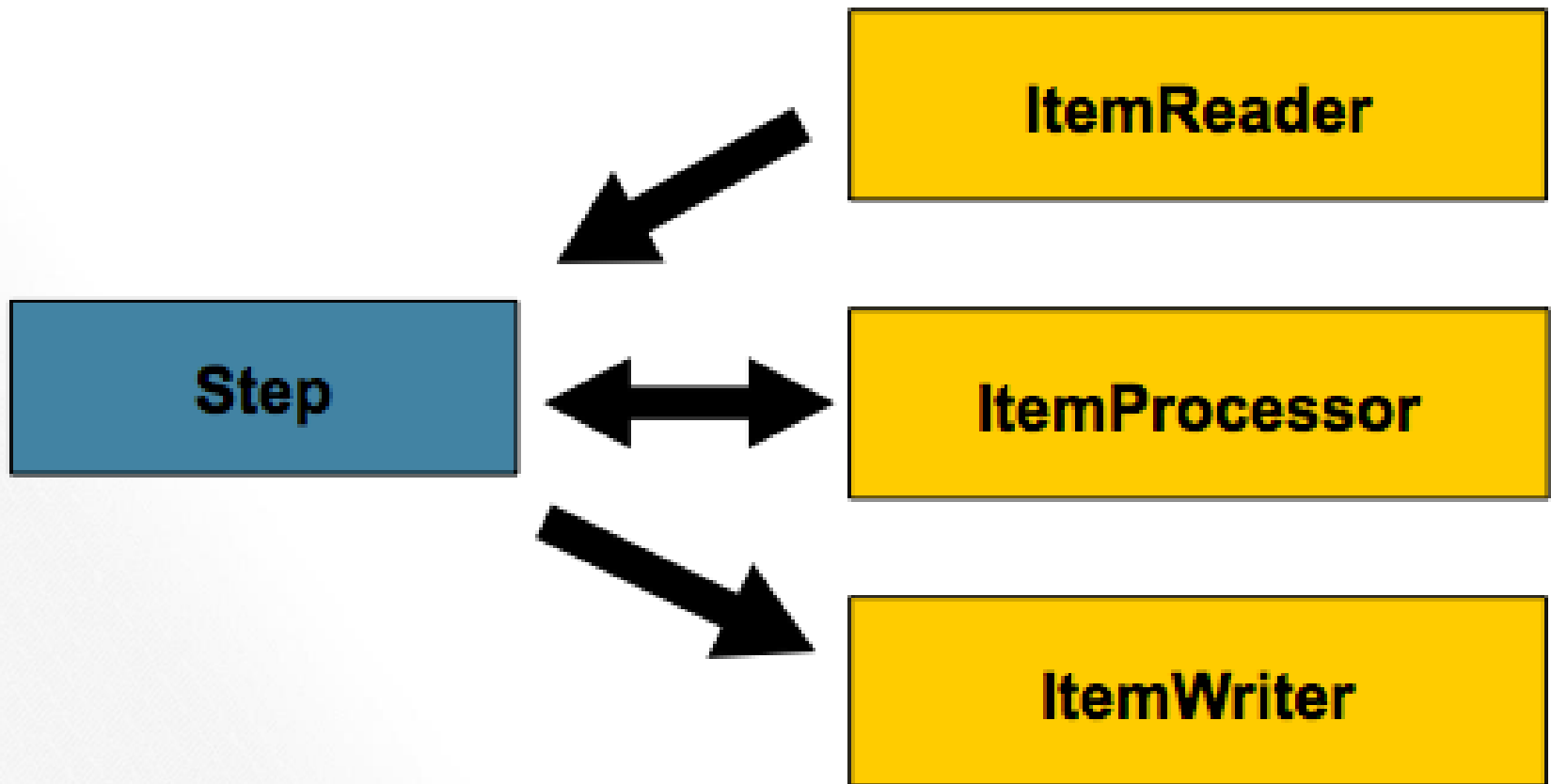
- **readCount** : nombre d'élément lu avec succès. (en lien avec ItemReader)
- **writeCount** : nombre d'élément écrit avec succès. (en lien avec ItemWriter)
- **commitCount** : nombre de transaction commité pour cette exécution (en lien avec le transactionManager)
- **rollbackCount** : nombre de rollback pour cette exécution (en lien avec le transactionManager)
- **readSkipCount** : nombre d'élément lu avec erreur, entrainant le passage à l'élément suivant. (en lien avec ItemReader)
- **writeSkipCount** : nombre d'élément écrit avec erreur, entrainant le passage à l'élément suivant. (en lien avec ItemWriter)
- **processSkipCount** : nombre d'élément traité avec erreur, entrainant le passage à l'élément suivant. (en lien avec ItemProcessor)
- **filterCount** : le nombre d'élément filtré lors du traitement (en lien avec ItemProcessor)

EXECUTIONCONTEXT

- + L'objet ExecutionContext représente un dictionnaire persistant dans lequel vous pourrez stocker toutes les informations qui vous sembleront utiles.
 - Par exemple, la dernière ligne lue dans un fichier
- + Vous avez une instance de cet objet par
 - JobExecution
 - StepExecution
- + Il fonctionne comme une Map<String, Object>
 - put(String, Object)
 - putLong(String, long)
 - putInt(String, int)
 - putDouble(String, double)
 - putString(String, String)
 - Object get(String)
 - long getLong(String)
 - long getLong(String, long default)
 - int getInt(String)
 - int getInt(String, int default)
 - double getDouble(String)
 - double getDouble(String, double default)
 - String getString(String)
 - String getString(String, String default)

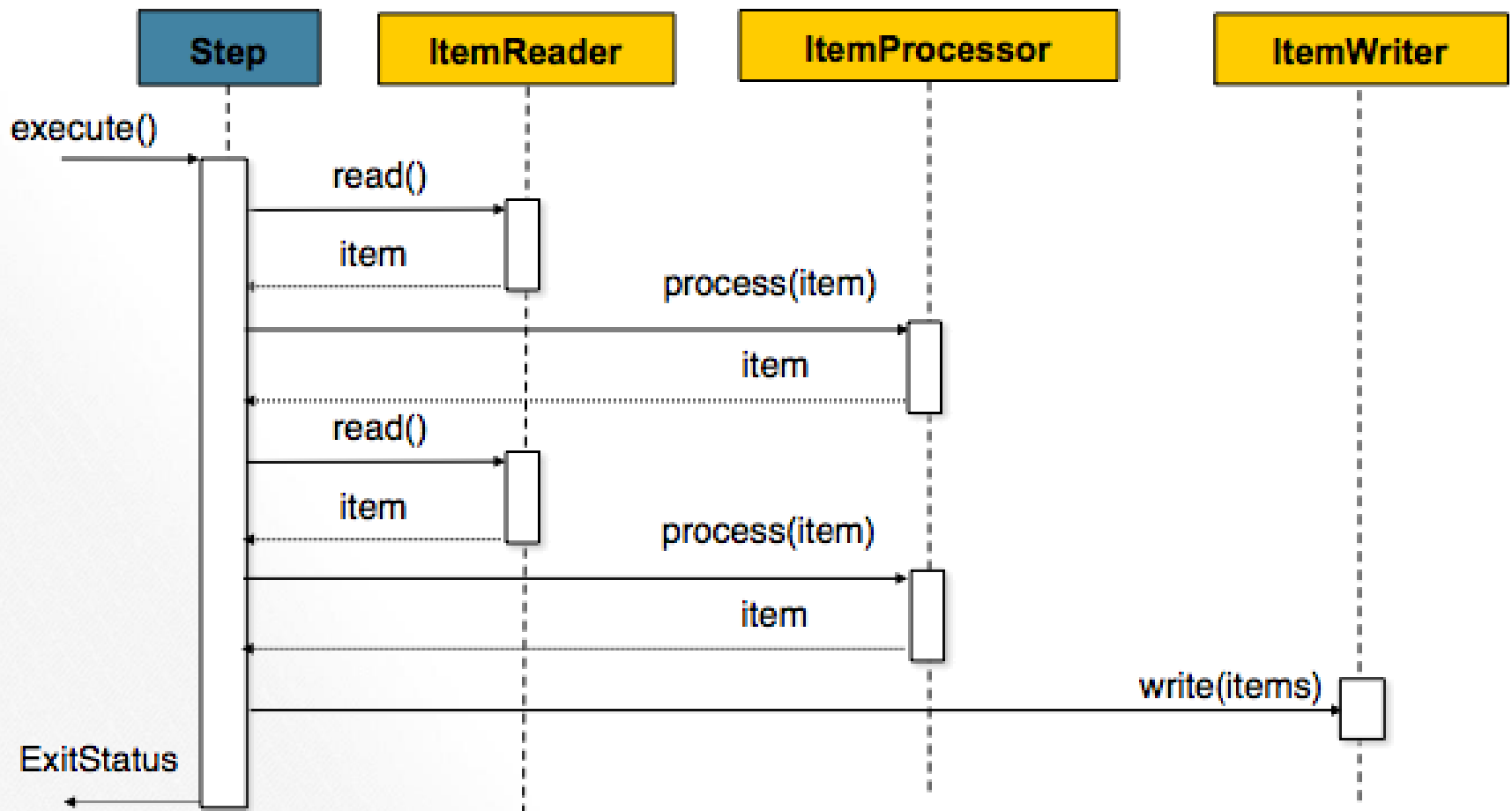
STEP

+ Fonctionnement d'un Step



STEP

+ Fonctionnement d'un Step



STEP

- + Un *Step* sera composé d'une *tasklet* qui aura
 - Un seul **ItemReader** : entrée des données
 - Lira une information à la fois
 - Retourne null quand il n'a plus rien à lire
 - Plusieurs implémentations disponibles
 - 0 ou Un **ItemProcessor** : traitement des données
 - Retourne l'élément à destination de l'ItemWriter
 - Si retourne null, l'élément ne sera pas traité par l'ItemWriter
 - Un seul **ItemWriter** : sortie des données
 - Récupère une liste d'item et les traite

STEP - ITEMREADER

+ ItemReader

```
package org.springframework.batch.item;

public interface ItemReader<T> {

    /**
     * Reads a piece of input data and advance to the next one. Implementations
     * <strong>must</strong> return <code>null</code> at the end of the input
     * data set. In a transactional setting, caller might get the same item
     * twice from successive calls (or otherwise), if the first call was in a
     * transaction that rolled back.
     *
     * @throws Exception if an underlying resource is unavailable.
     */
    public abstract T read() throws Exception, UnexpectedInputException,
                                                                    ParseException;

}
```

STEP - ITEMREADER

+ Vous trouverez les ItemReader suivants :

- ▶ **I** ItemReader<T>
 - > **G**^A AbstractItemCountingItemStreamItemReader<T>
 - G**^A AbstractItemStreamItemReader<T>
 - G** ItemReaderAdapter<T>
 - G** JmsItemReader<T>
 - G** ListItemReader<T>
 - G** MultiResourceItemReader<T>
 - G** OffsetItemReader<T>
- ▼ **I** ItemStreamReader<T>
 - ▼ **I** ResourceAwareItemReaderItemStream<T>
 - G** FlatFileItemReader<T>
 - G** StaxEventItemReader<T>

STEP - ITEMPROCESSOR

+ ItemProcessor








```
package org.springframework.batch.item;

/**
 * Interface for item transformation. Given an item as input, this interface
 * provides an extension point which allows for the application of business
 * logic in an item oriented processing scenario.
 * It should be noted that while it's possible to return a different type than
 * the one provided, it's not strictly necessary. Furthermore, returning null
 * indicates that the item should not be continued to be processed.
 */
public interface ItemProcessor<I, O> {

    /**
     * Process the provided item, returning a potentially modified or new item for
     * continued processing.
     * If the returned result is null, it is assumed that processing of the item
     * should not continue.
     *
     * @param item to be processed
     * @return potentially modified or new item for continued processing, null if
     * processing of the provided item should not continue.
     * @throws Exception
     */
    public abstract O process(I item) throws Exception;
}
```

STEP - ITEMPROCESSOR

+ Vous trouverez les ItemProcessor suivants :

-  `ItemProcessor<I, O>`
-  `CompositeItemProcessor<I, O>`
-  `ItemProcessorAdapter<I, O>`
-  `PassThroughItemProcessor<T>`
-  `UtilisateurItemProcessor`
-  `ValidatingItemProcessor<T>`
-  `new ItemProcessor() {...}<T, S>`

STEP - ITEMWRITER

+ ItemWriter

```
package org.springframework.batch.item;
















/**
 * <p>Basic interface for generic output operations. Class implementing this interface
 * will be responsible for serializing objects as necessary. Generally, it is
 * responsibility of implementing class to decide which technology to use for mapping
 * and how it should be configured. </p>
 *
 * <p>The write method is responsible for making sure that any internal buffers are
 * flushed. If a transaction is active it will also usually be necessary to discard
 * the output on a subsequent rollback. The resource to which the writer is sending
 * data should normally be able to handle this itself. </p>
 */
public interface ItemWriter<T> {

    /**
     * Process the supplied data element. Will not be called with any null items in normal
     * operation.
     *
     * @throws Exception if there are errors. The framework will catch the exception and
     * convert or rethrow it as appropriate.
     */
    public abstract void write(java.util.List<? extends T> items) throws Exception;

}
```

STEP - ITEMWRITER

+ Vous trouverez les ItemWriter suivants :

- ▼  ItemWriter<T>
 -  ^A AbstractItemStreamItemWriter<T>
 -  ClassifierCompositemWriter<T>
 -  CompositemWriter<T>
 -  HibernateItemWriter<T>
 -  IbatisBatchItemWriter<T>
 -  ItemWriterAdapter<T>
 -  JdbcBatchItemWriter<T>
 -  JmsItemWriter<T>
 -  JpItemWriter<T>
 -  MultiResourceItemWriter<T>
 -  PropertyExtractingDelegatingItemWriter<T>
- ▼  ResourceAwareItemWriterItemStream<T>
 -  FlatFileItemWriter<T>
 -  StaxEventItemWriter<T>

JOBREPOSITORY

- + Première étape lors du paramétrage des *Job*
- + Deux types de JobRepository
 - Mémoire
 - Simple, rapide pour faire des tests, ou si on ne s'intéresse pas aux différents états après les lancements de Job
 - En base de données
 - Nécessite l'installation de la base de données
 - Nécessite la définition d'une data source spécifique
 - Nécessite la mise en place de Job de purge
 - Permet de conserver longtemps toutes les informations relatives aux Job

JOBREPOSITORY - MEMOIRE

+ En mémoire :

```
<bean id="jobRepository"
  class="org.springframework.batch.core.repository.support.MapJobRepositoryFactoryBean">
  <!-- On lui associe son transaction manager directement -->
  <property name="transactionManager">
    <bean
      class="org.springframework.batch.support.transaction.ResourcelessTransactionManager" />
    </property>
  </bean>
```

+ Vous pouvez indiquer un autre id que `jobRepository` mais ce n'est pas conseillé

JOBREPOSITORY – EN BASE

+ Installer la base de données :

➤ Les scripts SQL sont à la racine du JAR
spring-batch-core

➤ Sélectionnez le scripte adapté à votre
infrastructure (DB2, Oracle, Mysql, ...)

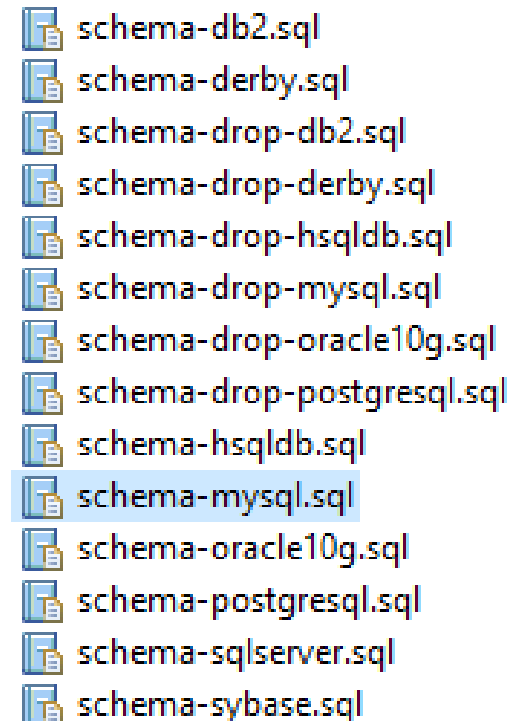
- Pensez à créer un schéma ou une
database

En MySQL :

```
create database `spring`;  
USE `spring`;
```

➤ Vous pouvez ajoutez un utilisateur dédié

- C'est mieux que d'utiliser l'accès
ROOT/SA



JOBREPOSITORY – EN BASE

+ Paramétrez votre Spring :

➤ 1 – Utilisez tous les namespace dans vos fichiers XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans
    xmlns="http://www.springframework.org/schema/batch"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/batch
        http://www.springframework.org/schema/batch/spring-batch.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">
```

Ici, batch
sera le
namespace
par défaut

JOBREPOSITORY – EN BASE

Paramétrez votre Spring - Suite

+ 2 – Déclarer le jobRepository

```
<job-repository id="jobRepository"
    data-source="dataSourceForJobRepository"
    transaction-manager="transactionManagerForJobRepository"
/>
```

Rappel :
batch est le
namespace
par défaut

+ 3 – Indiquer les contraintes transactionnelles

```
<aop:config>
  <aop:pointcut id="jobRepositoryTransactionPointcut"
    expression="execution(* org.springframework.batch.core.*Repository+.*(..))" />
  <aop:advisor advice-ref="txAdviceForJobRepository"
    pointcut-ref="jobRepositoryTransactionPointcut" />
</aop:config>

<tx:advice id="txAdviceForJobRepository"
  transaction-manager="transactionManagerForJobRepository">
  <tx:attributes><tx:method name="*" /></tx:attributes>
</tx:advice>
```

JOBREPOSITORY – EN BASE

Paramétrez votre Spring - Suite

+ 4 – Indiquer la datasource

```
<beans:bean id="dataSourceForJobRepository" destroy-method="close"
            class="org.apache.commons.dbcp2.BasicDataSource">
  <beans:property name="driverClassName" value="${db.job.repository.driver}" />
  <beans:property name="url" value="${db.job.repository.url}" />
  <beans:property name="username" value="${db.job.repository.login}" />
  <beans:property name="password" value="${db.job.repository.password}" />
</beans:bean>
```

+ Important :

- Ne pas oublier d'indiquer les informations associées à la base de données dans son fichier properties

```
<context:property-placeholder location="d.properties,d2.properties" />
```


JOBREPOSITORY – EN BASE

Paramétrez votre Spring - Suite

+ 5 – Déclarer le gestionnaire de transaction

```
<beans:bean id="transactionManagerForJobRepository"  
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
    <beans:property name="dataSource" ref="dataSourceForJobRepository" />  
</beans:bean>
```

+ Important :

- DataSource, TransactionManager, TransactionAdvice sont dédiés pour votre base de données Spring Batch

JOB LAUCHER

+ Il faut maintenant déclarer un JobLauncher

Rappel :
batch est le
namespace
par défaut

```
<beans:bean id="jobLauncher"  
  class="org.springframework.batch.core.launch.support.SimpleJobLauncher">  
  <beans:property name="jobRepository" ref="jobRepository" />  
</beans:bean>
```

+ Si vous souhaitez avoir la main sur la manière d'exécuter les Job, vous pouvez setter la propriété taskExecutor.

➤ Par exemple, pour lancer les Job de manière asynchrone

```
<beans:bean id="jobLauncher"  
  class="org.springframework.batch.core.launch.support.SimpleJobLauncher">  
  <beans:property name="jobRepository" ref="jobRepository" />  
  <beans:property name="taskExecutor">  
    <beans:bean class="org.springframework.core.task.SimpleAsyncTaskExecutor" />  
  </beans:property>  
</beans:bean>
```

JOB

- + Vous pouvez enfin déclarer vos Job et les Step qui le composent

```
<job id="ioSampleJob">
  <step id="step1">
    <tasklet>
      <chunk reader="itemReader"
              writer="itemWriter"
              commit-interval="2"/>
    </tasklet>
  </step>
</job>
```

Rappel :
batch est le
namespace
par défaut

- + Le *commit-interval* indique tous les combien d'éléments il faut générer une transaction. Ici on va lire 2 éléments via le reader, puis les passer au writer et réaliser une transaction.
- + Si vous avez plusieurs JobRepository ou si il n'a pas l'id jobRepository, vous pouvez ajouter l'attribut *job-repository* à votre balise job.

JOB

+ Rappel :

- un JobInstance = Job + Parametres
- Quand un Job s'exécute normalement, ⇔ status = COMPLETED, vous ne pourrez pas le relancer sans modifier les paramètres
 - Le cas échéant vous aurez un JobRestartException

+ Si vous ne voulez pas qu'un Job soit relancer, vous pouvez y ajouter l'attribut restartable="false"

- Que le Job se soit bien ou mal terminé, il ne sera pas relancé

+ Vous pouvez écouter les évènements associés à un Job via des objets de type JobExecutionListener

JOB

JobExecutionListener

+ Cette interface est composée de deux méthodes :

```
package org.springframework.batch.core;

/**
 * Provide callbacks at specific points in the lifecycle of a {@link Job}.
 * Implementations can be stateful if they are careful to either ensure thread
 * safety, or to use one instance of a listener per job, assuming that job
 * instances themselves are not used by more than one thread.
 */
public interface JobExecutionListener {

    /**
     * Callback before a job executes.
     *
     * @param jobExecution the current {@link JobExecution}
     */
    public abstract void beforeJob(JobExecution jobExecution);

    /**
     * Callback after completion of a job. Called after both both successful and
     * failed executions. To perform logic on a particular status, use
     * "if (jobExecution.getStatus() == BatchStatus.X)".
     *
     * @param jobExecution the current {@link JobExecution}
     */
    public abstract void afterJob(JobExecution jobExecution);

}
```

JOB

JobExecutionListener

- + Vous pouvez aussi réaliser une classe et utiliser les annotations
 - **@BeforeJob** : sur la méthode qui sera appelée avant l'exécution du Job
 - **@AfterJob** : sur la méthode qui sera appelée après l'exécution du Job

```
package fr.mon.projet.batch;

import org.springframework.batch.core.*;
import org.springframework.batch.core.annotation.*;

public class MonJobExecutionListenerAvecAnnotation {

    @BeforeJob
    public void avantLeJob(JobExecution jobExecution) {
        // Execution d'un code avant le depart du Job
    }

    @AfterJob
    public void apresLeJob(JobExecution jobExecution) {
        // Execution d'un code apres la fin du Job
        // Peut importe son status
    }
}
```

JOB

JobExecutionListener

+ L'ajout des listener se fait via la balise <listeners>

```
<job id="ioSampleJob">
  <step id="step1">
    <tasklet>
      <chunk reader="itemReader"
              writer="itemWriter"
              commit-interval="2"/>
    </tasklet>
  </step>
  <listeners>
    <listener ref="monListener"/>
  </listeners>
</job>
```

Rappel :
batch est le
namespace
par défaut

+ L'attribut **merge** de la balise *listeners* permet d'indiquer, dans le cas d'un héritage, si il faut fusionner les listes de listeners ou les écraser.

HÉRITAGE ENTRE JOB

- + Tout comme n'importe quel bean spring, vous pouvez faire usage des attributs `parent`, `abstract` pour mettre en place de l'héritage entre vos Job

```
<job id="absJob" abstract="true">
  <step id="step1">
    <tasklet>
      <chunk reader="itemReader" writer="itemWriter" />
    </tasklet>
  </step>
</job>
```

```
<job id="ioSampleJob" parent="absJob">
  <listeners>
    <listener ref="monListener"/>
  </listeners>
</job>
```

Rappel :
batch est le
namespace
par défaut

VALIDATION DES PARAMÈTRES DU JOB

JobParametersValidator

- + Si vous le souhaitez, vous pouvez ajouter un validateur de paramètre pour votre Job
 - Via la classe DefaultJobParametersValidator
 - Via une classe qui implémente JobParametersValidator
- + Depuis v3 de spring-batch

```
<job id="ioSampleJob">
  <step id="step1">
    <tasklet>
      <chunk reader="itemReader"
        writer="itemWriter"
        commit-interval="2"/>
    </tasklet>
  </step>
  <validator ref="parametersValidator"/>
</job>
```

Rappel :
batch est le
namespace
par défaut

HÉRITAGE ENTRE STEP

- + Tout comme n'importe quel bean spring, vous pouvez faire usage des attributs `parent`, `abstract` pour mettre en place de l'héritage entre Step

```
<job id="ioSampleJob">
  <step id="absStep" abstract="true">
    <tasklet>
      <chunk commit-interval="2"/>
    </tasklet>
  </step>
  <step id="step1" parent="absStep">
    <tasklet>
      <chunk reader="itemReader"
              writer="itemWriter" />
    </tasklet>
  </step>
</job>
```

Rappel :
batch est le
namespace
par défaut

OPTIONS SUR TASKLET

start-limit

- + Vous pouvez préciser le nombre de fois qu'un Step peut être lancé.
 - Par exemple, limiter à 1
- + **Attention** : si vous tentez de dépasser le nombre limite, une exception sera lancée.

```
<job id="ioSampleJob">
  <step id="step1">
    <tasklet start-limit="2">
      <chunk reader="itemReader"
              writer="itemWriter" />
    </tasklet>
  </step>
</job>
```

Rappel :
batch est le
namespace
par défaut

OPTIONS SUR TASKLET

allow-start-if-complete

- + Vous pouvez préciser si un Step doit toujours être lancée.
 - Par exemple, pour nettoyer des ressources
- + **Attention** : par défaut si un Step est COMPLETED, il n'est pas relancé

```
<job id="ioSampleJob">
  <step id="step1">
    <tasklet allow-start-if-complete="true">
      <chunk reader="itemReader"
              writer="itemWriter" />
    </tasklet>
  </step>
</job>
```

Rappel :
batch est le
namespace
par défaut

STEP

StepExecutionListener

+ Vous pouvez écouter les événements associés à un Step

```
package org.springframework.batch.core;
/**
 * Listener interface for the lifecycle of a {@link Step}.
 */
public interface StepExecutionListener extends StepListener {
    /**
     * Initialize the state of the listener with the {@link StepExecution} from
     * the current scope.
     *
     * @param stepExecution
     */
    public abstract void beforeStep(StepExecution stepExecution);

    /**
     * Give a listener a chance to modify the exit status from a step. The value
     * returned will be combined with the normal exit status using
     * {@link ExitStatus#and(ExitStatus)}.
     *
     * Called after execution of step's processing logic (both successful or
     * failed). Throwing exception in this method has no effect, it will only be
     * logged.
     *
     * @return an {@link ExitStatus} to combine with the normal value. Return
     * null to leave the old value unchanged.
     */
    public abstract ExitStatus afterStep(StepExecution stepExecution);
}
```

STEP

StepExecutionListener

- + Vous pouvez aussi réaliser une classe et utiliser les annotations
 - **@BeforeStep** : sur la méthode qui sera appelée avant l'exécution du Step
 - **@AfterStep** : sur la méthode qui sera appelée après l'exécution du Step

```
package fr.mon.projet.batch;

import org.springframework.batch.core.*;
import org.springframework.batch.core.annotation.*;

public class MonStepExecutionListenerAvecAnnotation {

    @BeforeStep
    public void avantLeStep(StepExecution stepExecution) {
        // Execution d'un code avant le depart du Step
    }

    @AfterStep
    public ExitStatus apresLeStep(StepExecution stepExecution) {
        // Execution d'un code apres la fin du Step
        // Peut importe son status
        return null; // Afin de ne pas modifier le vrai status du Step
    }
}
```

STEP

StepExecutionListener

+ L'ajout des listener se fait via la balise <listeners>

```
<job id="ioSampleJob">
  <step id="step1">
    <tasklet>
      <chunk reader="itemReader"
              writer="itemWriter"
              commit-interval="2"/>
    </tasklet>
    <listeners>
      <listener ref="monListener"/>
    </listeners>
  </step>
</job>
```

Rappel :
batch est le
namespace
par défaut

+ Rappel : vous pouvez, dans le cas d'un héritage, faire usage de l'attribut **merge** de la balise *listeners*

➤ Elle permet de **fusionner** ou **écraser** les listes de listeners.

LANCEMENT D'UN JOB

A la main

+ Vous pouvez créer votre propre classe Java de lancement de Job

➤ Un main

- Vous chargez votre contexte Spring
- Vous récupérez votre bean jobLauncher
- Vous créez vos paramètres (JobParameters)
La création de JobParameters se fait à l'aide du JobParametersBuilder
- Et vous lancez l'exécution du job en récupérant le bean qui le représente

LANCEMENT D'UN JOB

```
package com.banque.batch;

import org.apache.logging.log4j.*;
import org.springframework.batch.core.*;
import org.springframework.context.support.*;

public final class LanceurDeJob {
    private static final Logger LOG = LogManager.getLogger();

    public static void main(String[] args) {
        LanceurDeJob.LOG.debug("-- Debut du programme --");

        final String[] springConfig = { "batch/*-context.xml", "*-context.xml" };
        ClassPathXmlApplicationContext context = null;
        try {
            context = new ClassPathXmlApplicationContext(springConfig);
            JobLauncher jobLauncher = context.getBean("jobLauncher", JobLauncher.class);
            Job job = context.getBean("jobExample", Job.class);
            JobExecution execution = jobLauncher.run(job, new JobParameters());
            LanceurDeJob.LOG.printf(Level.INFO, "Fin du Job Status : %s", execution.getStatus());
        } catch (Exception e) {
            LanceurDeJob.LOG.error("Erreur: ", e);
        } finally {
            if (context != null) {
                context.close();
            }
        }
        LanceurDeJob.LOG.debug("-- Fin du programme --");
    }
}
```

LANCEMENT D'UN JOB

CommandLineJobRunner

- + Vous pouvez utiliser (ou hériter) de la classe `org.springframework.batch.core.launch.support.CommandLineJobRunner`
- + Cette classe de Spring batch contient un main, il suffira de lui faire passer l'ensemble des informations nécessaires au lancement :
 - En **premier** paramètre, le **jobPath** : votre fichier de configuration spring
 - En **second** paramètre, le **jobName** : l'id de votre bean Job
 - En **troisième** paramètre, les **jobParameters** : l'ensemble de vos paramètres.
 - -restart: (optional) si le job a échoué ou est arrêté, relancera le job (repartira du JobExecution)
 - -next: (optional) si vous avez mis en place un enchainement de job à travers un JobParametersIncrementer

LANCEMENT D'UN JOB

CommandLineJobRunner

- + **Important** : n'oubliez de configurer correctement votre **path** et **classpath** avant de lancer votre commande
 - PATH : doit contenir un chemin vers un JDK
 - CLASSPATH : doit contenir l'ensemble des librairies, chemins de ressources nécessaire à l'exécution de votre processus Java
 - Cette option peut être gérée dynamiquement lors du lancement avec `-cp` ou `-classpath`

+ Exemple d'une ligne de lancement

```
java CommandLineJobRunner testJob.xml testJob  
    schedule.date=2008/01/24 vendor.id=3902483920
```

+ Les paramètres du job sont présentés sous la forme `clef=valeur`

RÉSULTAT DU LANCEMENT D'UN JOB

- + La classe `org.springframework.batch.core.ExitStatus` représente les codes retours de vos Step et Job
 - Cette classe encapsule les informations

- + Dans le cas de l'utilisation du `CommandLineJobRunner`
 - ~ `System.exit(0)` si tout ce passe bien
 - ~ `System.exit(1)` si une erreur générique survient
 - ~ `System.exit(2)` si le lanceur de Job a un problème
 - Si vous voulez obtenir plus de codes retours, vous pouvez brancher sur l'objet `CommandLineJobRunner` un objet de type `ExitCodeMapper`
 - Il suffit de le déclarer dans un de vos fichiers Spring

JOBEXPLORER

- + Spring vous permet de créer des objets susceptibles d'explorer vos Job
- + Cet objet devra implémenter l'interface `org.springframework.batch.core.explore.JobExplorer`
 - `public abstract List<JobInstance> getJobInstances(String jobName, int start, int count);`
 - `public abstract JobExecution getJobExecution(Long executionId);`
 - `public abstract StepExecution getStepExecution(Long jobExecutionId, Long stepExecutionId);`
 - `public abstract JobInstance getJobInstance(Long instanceId);`
 - `public abstract List<JobExecution> getJobExecutions(JobInstance jobInstance);`
 - `public abstract Set<JobExecution> findRunningJobExecutions(String jobName);`

JOBEXPLORER

- + De base, vous pouvez obtenir un JobExplorer en le demandant à sa factory.

```
<bean id="jobExplorer"  
  class="org.springframework.batch.core.explore.support.JobExplorerFactoryBean">  
    <property name="datasource" ref="dataSourceForJobRepository" />  
</bean>
```

- + N'oubliez pas de le lier à votre datasource dédiée à Spring batch.
- + Le JobExplorer peut être assimilé à un JobRepository en read-only.

JOBREGISTRY

- + Cet objet permet d'avoir une liste de l'ensemble des Job

```
<bean id="jobRegistry"  
  class="org.springframework.batch.core.configuration.support.MapJobRegistry" />
```

- + Il s'utilise rarement tout seul, mais plutôt avec un JobRegistryBeanPostProcessor ou un JobOperator
- + Le JobRegistryBeanPostProcessor permet de référencer automatiquement les Job dans le JobRegistry

```
<bean id="jobRegistryBeanPostProcessor"  
  class="org.springframework.batch.core.  
    configuration.support.JobRegistryBeanPostProcessor">  
  <property name="jobRegistry" ref="jobRegistry"/>  
</bean>
```

JOBOPERATOR

+ Le JobOperator permet de manipuler les Job référencés

```
<bean id="jobOperator"  
    class="org.springframework.batch.core.launch.support.SimpleJobOperator">  
    <property name="jobExplorer" ref="jobExplorer" />  
    <property name="jobRepository" ref="jobRepository" />  
    <property name="jobRegistry" ref="jobRegistry" />  
    <property name="jobLauncher" ref="jobLauncher" />  
</bean>
```

+ Méthodes associées :

- 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()

JOBOPERATOR

- + Le JobOperator est une forme de JobLaucher mais plus intelligent.
- + La méthode **startNextInstance** va
 - relancer le Job en modifiant automatiquement les valeurs des paramètres si ce dernier ne s'est pas terminé correctement
 - Lancé le Job suivant si le Job courant s'est terminé correctement
- + Dans tous les cas cette méthode fera usage du JobParametersIncrementer pour Fabriquer de nouveaux paramètres pour le Job

JOBPARAMETERSINCREMENTER

- + Cet objet permet de générer des JobParameters
- + Il s'utilise avec le JobParametersBuilder qui permet de fabriquer des JobParameter
- + Par exemple :

```
package com.banque.batch;

import org.springframework.batch.core.*;

public class SimpleJobParameter implements JobParametersIncrementer {

    public JobParameters getNext(JobParameters aP) {
        long id = 1L;
        // Si pas de parametres = premier lancement du job
        if (aP == null || aP.isEmpty()) {
            // On ne fait rien id = 1
        } else {
            // Sinon on incremente l'id
            id = aP.getLong("run.id", 1L) + 1L;
        }
        return new JobParametersBuilder().addLong("run.id", id).toJobParameters();
    }
}
```

JOBPARAMETERSINCREMENTER

+ Il se lie au Job lors de sa déclaration

```
<beans:bean id="simpleJobParameter"
            class="com.banque.batch.SimpleJobParameter" />

<job id="monSuperJob" incrementer="simpleJobParameter">
    ...
</job>
```