Spring Batch

Francisco Philip
fphilip@houseware.es

¿Qué es Spring Batch?

Es un framework ligero y completo diseñado para facilitar el desarrollo de aplicaciones batch robustas.

Características

- Gestión de transacciones
- Procesamiento basado Chunk (trozos)
- Entrada y Salida declarativa
- Control Start / Stop / Restart
- Política de reintentos / saltos
- Interfaz de administración Web

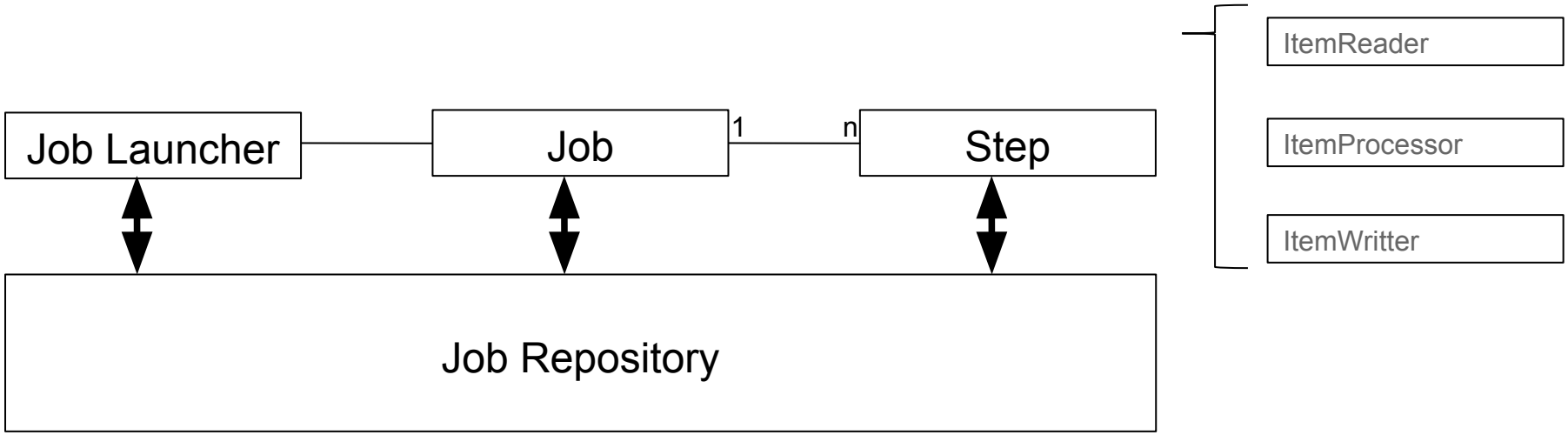
Procesamiento por lotes

Ejecución de un programa sin el control o supervisión directa del usuario.

Se utiliza en tareas repetitivas sobre grandes conjuntos de información.

En un sistema por lotes existe un gestor de trabajos encargado de reservar y asignar los recursos de las máquinas a las tareas que hay que ejecutar.

Conceptos básicos



- Un **Job** está formado por uno o más **Steps**
- Un Job se ejecuta mediante un **Job Laucher**
- El estado de la ejecución se almacena en el **Job Repository**

Job

- Define el **job** y como será ejecutado
  - Especifica la secuencia de **steps** que lo forman
  - Configuración:
    - restartable

JobInstance – JobParameter - JobExecution

- **JobInstance** son cada una de las ejecuciones lógicas del job. Permite organizar los detalles de la ejecución.

JobInstance = Job + **JobParameters**

- **JobExecution** es cada uno de los intentos de ejecutar un job.

Una ejecución puede terminar en error o éxito, pero el JobInstance correspondiente no se considera completa hasta que no finalice con éxito.

JobInstance y JobExecution

Tenemos un job que se ejecuta a final de mes para generar un informe.

La ejecución del job a fin de mes se representa mediante **JobInstance**.

Cada una de las ejecuciones de la instancia del Job hasta que se completa se representa mediante **JobExecution**.

JobParameters

¿Cómo se distingue una JobInstance de otra?

Por los parámetros utilizados para lanzar el Job

```
Map<String, JobParameter> params = new HashMap<String, JobParameter>();
params.put(“running.date”, new JobParameter(new java.util.Date(),true));

JobParameters parameters = new JobParameter(params);
```

JobParameters

Se admiten parámetros de los siguientes tipos:

- String
- Double
- Long
- java.util.Date

Los parámetros pueden ser **identificadores** (forman parte de la clave del JobInstance) o **informativos**.

Step

- Representa una fase secuencial de un **Job**.
- Un **step** contiene toda la información necesaria.
- Un step puede cargar datos de un fichero a la base de datos; o generar un fichero a partir de la base de datos, ….
- Al igual que **Job** se registran las ejecuciones del step en **StepExecution**

StepExecution

- Representa cada uno de los intentos de ejecutar un **Step**.
- Se crea una instancia de **StepExecution** cada vez que se ejecuta un Step.

ExecutionContext

- Representa de una colección de pares de clave/valor, que son persistidos y gestionados por el framework; para permitir a los desarrolladores un lugar donde almacenar información sobre un **StepExecution** o **JobExecution**

JobRepository

- Es el mecanismo de persistencia para todos los elementos descritos.
- Proporciona operaciones CRUD para **JobLauncher**, **Job** y **Step**.
- Cuando se lanza un Job, se obtiene un **JobExecution** del repositorio y durante su ejecución, se persiste junto con las instancias de **StepExecution**.

JobLauncher

- Representa un interfaz para lanzar un **Job** con un conjunto de **JobParameter**

```
public interface JobLauncher {

    public JobExecution run(Job job, JobParameters jobParameters)
                throws JobExecutionAlreadyRunningException, JobRestartException;
}
```

JobRepository
Esquema BBDD

BATCH_JOB_INSTANCE

BATCH_JOB_EXECUTION

BATCH_JOB_EXECUTION_PARAMS

BATCH_JOB_EXECUTION_CONTEXT

BATCH_STEP_EXECUTION

BATCH_STEP_EXECUTION_CONTEXT

Hello World!
Spring Batch

Hello World!

Configuración Base

- Configuración del base de datos para test.
- Configuración de la Transaccionalidad

```
<bean id="transactionManager"
      class="org.springframework.batch.support.transaction.ResourcelessTransactionManager"/>

<jdbc:embedded-database id="dataSource" type="H2" >
      <jdbc:script execution="INIT"
            location="classpath:org/springframework/batch/core/schema-drop-h2.sql"/>
       <jdbc:script execution="INIT"
            location="classpath:org/springframework/batch/core/schema-h2.sql"/>
</jdbc:embedded-database>
```

Hello World!

Configuración Base

- Configuración del repositorio
- Configuración del Launcher

```
<bean id="jobRepository"
      class="org.springframework.batch.core.repository.support.JobRepositoryFactoryBean">
      <property name="dataSource" ref="dataSource"/>
      <property name="transactionManager" ref="transactionManager"/>
      <property name="databaseType" value="h2"/>
</bean>

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

Hello World!

Configuración Job

```
<bean id="helloTask" class="houseware.learn.spring.springbatch.case01.HelloWorldTask"/>
<batch:job id="helloWorldJob">
        <batch:step id="helloWorldJobStep1">
                <batch:tasklet ref="helloTask"/>
        </batch:step>
</batch:job>

<bean id="parametersTask" class="houseware.learn.spring.springbatch.case01.ParametersTask"/>
<batch:job id="parametersTaskJob">
        <batch:step id="parametersTaskJobStep1">
                <batch:tasklet ref="parametersTask"/>
        </batch:step>
</batch:job>
```

Hello World!

Ejecución del Job

```
JobParameters jobParameters = new JobParametersBuilder()
        .addString("user", "Francisco")
        .addDate("date", new Date())
        .toJobParameters();

getJobLauncher().run(parametersTaskJob, jobParameters);
```

Entrada / Salida
(lectura y escritura)

Step

- Procesamiento por conjunto (**chunks**)
  - Lee un dato (**ItemReader**), lo procesa (**ItemProcessor**) y lo agrega.
  - Cuando el número de datos leídos es igual al *commit-interval*; el conjunto (chunk) es escrito (**ItemWriter**) y se realiza commit de la transacción.

Configuración Step

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

Lectura

Lectura/Procesamiento/Escritura

- ItemReader
- ItemProcessor
- ItemWriter

Lectura de datos ItemReader

```
public interface ItemReader<T> {

    T read() throws Exception, UnexpectedInputException,
   ParseException;

}
```

Ficheros de texto
(delimitado o posiciones fijas)

Lectura de ficheros de texto
FlatFileItemReader

- El procesamiento de los registros de entrada se encapsula en la propiedad **lineMapper** del FlatFileItemReader.

Lectura de ficheros de texto
FlatFileItemReader

- Hay varias implementaciones de la interfaz **LineMapper**:
  - **org.springframework.batch.item.file.mapping.PassThroughLineMapper** : Permite pasar el String leído directamente en lugar del objeto mapeado.
  - **org.springframework.batch.item.file.mapping.DefaultLineMapper** : Procesar el String leído en dos fases: tokenizer y generar un objeto de negocio con la información.
  - **org.springframework.batch.item.file.mapping.PatternMatchingCompositeLineMapper** : Permite procesar múltiples tipos de registros del mismo fichero.

Lectura de ficheros de texto
PassThroughLineMapper

```
<bean id="countriesCvsFileItemReader" class="org.springframework.batch.item.file.
    FlatFileItemReader">
  <property name="resource" value="classpath:countries.txt" />
  <property name="linesToSkip" value="1"/>
  <property name="lineMapper">
     <bean class="org.springframework.batch.item.file.mapping.PassThroughLineMapper">
     </bean>
  </property>
</bean>
```

Lectura de ficheros de texto
DefaultLineMapper [DelimitedLineTokenizer]

```
<bean id="countriesCvsFileItemReader" class="org.springframework.batch.item.file.
   FlatFileItemReader">
 <property name="resource" value="classpath:countries.txt" />
 <property name="linesToSkip" value="1"/>
 <property name="lineMapper">
    <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
      <property name="lineTokenizer">
        <bean
          class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
          <property name="names" value="country,isoCode" />
          <property name="delimiter" value=";"/>
       </bean>
      </property>
      <property name="fieldSetMapper">
        <bean
          class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper">
          <property name="prototypeBeanName" value="countryModel" />
        </bean>
      </property>
    </bean>
  </property>
</bean>
```

Lectura de ficheros de texto
DefaultLineMapper [FixedLengthTokenizer]

```
<property name="lineTokenizer">
    <bean       class="org.springframework.batch.item.file.transform.
   FixedLengthTokenizer">
        <property name="names" value="country,isoCode" />
        <property name="columns" value="1-20, 21-23"/>
    </bean>
</property>
```

Lectura de ficheros de texto
PatternMatchingCompositeLineMapper

```
<bean id="countriesCvsFileItemReader" class="org.springframework.batch.item.file.
    FlatFileItemReader">
  <property name="resource" value="classpath:countries.txt" />
  <property name="linesToSkip" value="1"/>
  <property name="lineMapper" ref="mapper">

</bean>

<bean id="mapper"
class="org.springframework.batch.item.file.mapping.PatternMatchingCompositeLineMapper">
  <property name="tokenizers">
    <map>
      <entry key="COUNTRY*" value-ref="countryTokenizer" />
      <entry key="CITY*" value-ref="cityTokenizer" />
    </map>
  </property>
  <property name="fieldSetMappers">
    <map>
      <entry key="COUNTRY*" value-ref="countryMapper" />
      <entry key="CITY*" value-ref="cityMapper" />
    </map>
  </property>
</bean>
```

Lectura de ficheros XML

Lectura de ficheros XML
StaxEventItemReader

- Para procesar XML se necesita:
  - El elemento raíz del fragmento que constituye el objeto a ser mapeado
  - El resource a leer (fichero de entrada)
  - UnMarshaller que hará el mapeo del XML al Objeto

Lectura de ficheros XML
StaxEventItemReader

Queremos cargar los actos obtenidos del siguiente fichero XML

```
<response>
  <header module="QueryNewLlistaAG2" operation="getNewLlistaAG" session=""/>
    <body idioma="CA">
      <resultat>
        <info>...</info>
        <actes>
          <acte>
            <id>99400149076</id>
            <nom>
            Visita dinamitzada per a escoles a la Sagrada Família - Museu Temple Expiatori
            </nom>
          </acte>
        <actes>
```

Lectura de ficheros XML

StaxEventItemReader

Documento XML

```
<acte>
   <id>99400149076</id>
   <nombre>
            Visita dinamitzada
per a escoles a la Sagrada
Família - Museu Temple Expiatori
   </nombre>
   <fecha>2012-09-30T10:00:
00</fecha>
</acte>
```

El elemento **id** del documento XML se guarda en la propiedad **id**

El elemento **nombre** del documento XML se guarda en la propiedad **nom**

El elemento **fecha** del documento XML no lo vamos a procesar

Java Bean

```
package model;
public class ActeModel {

  private String id;
  private String nom;


  public void setId() {

  }

  public void setNom() {

  }


}
```

Lectura de ficheros XML
StaxEventItemReader

```
<bean id="itemReader" class="org.springframework.batch.item.xml.
StaxEventItemReader">
    <property name="fragmentRootElementName" value="acte" />
    <property name="resource" value="data/input.xml" />
    <property name="unmarshaller" ref="acteMarshaller" />
</bean>

<bean id="acteMarshaller"
      class="org.springframework.oxm.xstream.XStreamMarshaller">
    <property name="aliases">
        <util:map>
            <entry key="acte" value="model.ActeModel"/>
        </util:map>
    </property>
    <property name="fieldAliases">
        <util:map>
            <entry key="model.ActeModel.nom" value="nombre"/>
        </util:map>
    </property>
    <property name="omittedFields">
        <util:map>
            <entry key="model.ActeModel" value="fecha"/>
        </util:map>
    </property>
</bean>
```

Lectura de ficheros XML
StaxEventItemReader

- Spring OXM => Añadir dependencia a Maven
  - groupId: org.springframework
  - artifactId: spring-oxm
  - version: La versión de Spring

Lectura de base de datos

Lectura de Base de datos
JdbcCursorItemReader

```
<bean id="itemReader" class="org.springframework.batch.item.
database.JdbcCursorItemReader">
    <property name="datasource" ref="dataSource" />
    <property name="sql" value="SELECT ID, NAME FROM DATA" />
    <property name="rowMapper">
      <bean class="com.mypackage.MyRowMapper"/>
    </property>
</bean>
```

org.springframework.jdbc.core.RowMapper

```
public interface RowMapper {

   public Object mapRow(ResultSet rs, int rowNum) throws java.sql.
SQLException;
}
```

Lectura de Base de datos
JdbcCursorItemReader

```
public class MyRowMapper implements RowMapper {

   public Object mapRow(ResultSet rs, int rowNum) throws java.sql.
SQLException {

      MyBean bean = new MyBean();
      // asignación variables rs.getString("COL_NAME");

      return bean;

   }
}
```

Procesamiento

Procesamiento de Datos ItemProcessor

```
public interface ItemProcessor<I, O> {

    O process(I item) throws Exception;

}
```

Usos:

- Transformar de un tipo de datos o otro
- Filtrar que datos no queremos que vayan al ItemWriter: Retornaremos **null**

Escritura

Escritura de Datos ItemWriter

```
public interface ItemWriter<T> {

    void write(List<? extends T> items) throws Exception;

}
```

Escritura de Ficheros FlatFileItemWriter

- Define una propiedad, shouldDeleteIfExists para eliminar el fichero a escribir antes de empezar. Sino, empezará a escribir en la última posición correcta.
- De la misma forma que la propiedad **lineMapper** indicaba como procesar las líneas de entrada, tenemos el **lineAggregator** para controlar el proceso de escritura.

Escritura de Ficheros FlatFileItemWriter

- Hay varias implementaciones de la interfaz **LineAggregator**:
  - **org.springframework.batch.item.file.transform.PassThroughLineMapper** : Llama al método . toString() del objeto.
  - **org.springframework.batch.item.file.transform.DelimitedLineAggregator** : Escribe un fichero en formato delimitado.
  - **org.springframework.batch.item.file.transform.FormattedLineAggregator** : Escribe un fichero en formato fijo.

Escritura de Datos
PassThroughLineMapper

```
<bean id="itemWriter" class="org.springframework.batch.item.file.
   FlatFileItemWriter">
     <property name="resource" ref="outputResource" />
     <property name="lineAggregator">
         <bean class="org.springframework.batch.item.file.
   transform.PassThroughLineAggregator">

         </bean>
     </property>
</bean>
```

Escritura de Datos
DelimitedLineAggregator

```
<bean id="itemWriter" class="org.springframework.batch.item.file.
   FlatFileItemWriter">
     <property name="resource" value="file:c:/outputResource.txt"
   />
     <property name="lineAggregator">
         <bean class="org.springframework.batch.item.file.
   transform.DelimitedLineAggregator">
             <property name="delimiter" value=","/>
             <property name="fieldExtractor">
                 <bean class="org.springframework.batch.item.file.
   transform.BeanWrapperFieldExtractor">
                     <property name="names" value="name,credit"/>
                 </bean>
             </property>
         </bean>
     </property>
</bean>
```

Escritura de Datos
FormatterLineAggregator

```
<bean id="itemWriter" class="org.springframework.batch.item.file.
   FlatFileItemWriter">

    <property name="resource" ref="outputResource" />

    <property name="lineAggregator">

        <bean class="org.springframework.batch.item.file.transform.
   FormatterLineAggregator">

            <property name="fieldExtractor">

                <bean class="org.springframework.batch.item.file.transform.
   BeanWrapperFieldExtractor">

                    <property name="names" value="name,credit" />

                </bean>

            </property>

            <property name="format" value="%-9s%-2.0f" />

        </bean>

    </property>

</bean>
```

Escritura de Ficheros
FlatFileItemWriter

Podemos escribir registros al inicio y al final del fichero con información del proceso.

Se implementa mediante dos **callbacks**, que se definen en el bean writer mediante las propiedades *headerCallback* y *footerCallback*

Escritura de Ficheros
FlatFileItemWriter

```
public interface FlatFileHeaderCallback {
    /**
     * Write contents to a file using the supplied {@link Writer}. It is not
     * required to flush the writer inside this method.
     */
    void writeHeader(Writer writer) throws IOException;
}
public interface FlatFileFooterCallback {
    /**
     * Write contents to a file using the supplied {@link Writer}. It is not
     * required to flush the writer inside this method.
     */
    void writeFooter(Writer writer) throws IOException;
}
```

Escritura de Ficheros
FlatFileItemWriter

```
<bean id="itemWriter" class="org.springframework.batch.item.file.
   FlatFileItemWriter">

    <property name="headerCallback" ref="headerCallback" />

    <property name="footerCallback" ref="footerCallback" />

</bean>
```

Escritura de Ficheros
ItemStream

```
public interface ItemStream {

    /**
     * Open the stream for the provided {@link ExecutionContext}.
     *
     * @throws IllegalArgumentException if context is null
     */
    void open(ExecutionContext executionContext) throws ItemStreamException;

    /**
     * Indicates that the execution context provided during open is about to be saved. If
any state is remaining, but
     * has not been put in the context, it should be added here.
     *
     * @param executionContext to be updated
     * @throws IllegalArgumentException if executionContext is null.
     */
    void update(ExecutionContext executionContext) throws ItemStreamException;

    /**
     * If any resources are needed for the stream to operate they need to be destroyed here.
Once this method has been
     * called all other methods (except open) may throw an exception.
     */
    void close() throws ItemStreamException;
}
```

Escritura en Base de datos

Escritura en Base de datos FlatFileItemWriter

```
 <bean id="countriesDBItemWriter"
        class="org.springframework.batch.item.database.JdbcBatchItemWriter">
        <property name="dataSource" ref="dataSource" />
        <property name="sql">
         <value>
            <![CDATA[
        insert into COUNTRIES(COUNTRY, ISO_CODE)
                        values (:country, :isoCode)
]]>
         </value>
        </property>
        <!-- It will take care matching between object property and sql name parameter -->
        <property name="itemSqlParameterSourceProvider">
                <bean
     class="org.springframework.batch.item.database.BeanPropertyItemSqlParameterSourceProvider" />
        </property>
  </bean>
```

Envolviendo la escritura de Datos
Patrón Delegate

```
public class CompositeItemWriter<T> implements ItemWriter<T> {

    ItemWriter<T> itemWriter;

    public void write(List<? extends T> items) throws Exception {
        //Lógica de negocio
       itemWriter.write(item);
    }

    public void setDelegate(ItemWriter<T> itemWriter){
        this.itemWriter = itemWriter;
    }
}
```

Envolviendo la escritura de Datos
Patrón Delegate

Al utilizar un writer delegado, hemos de indicar en el **step** el stream para que se encargue del proceso de apertura, cierre y actualización del estado de procesamiento

```
<batch:chunk reader="myReader" writer="myWriter"
             commit-interval="10">
          <batch:streams>
            <batch:stream ref="delegateWriter"/>
          </batch:streams>
 </batch:chunk>
```

Procesamiento

Gestionando el procesamiento de Steps

- Limitar el número de ejecuciones de un Step
- Volver a ejecutar un Step
- Saltar errores de lectura
- Reintento de lectura/escritura
- Rollback
- Listeners

Limitando el número de ejecuciones de un Step

- Si queremos que un step se ejecuten una única vez porque *invalida un recurso* que se ha corregir a mano, utilizaremos el atributo **start-limit=”1”** del tasklet.

Si se intenta volver a ejecutar el step, se lanzará una excepción; lo que requerirá intervención del operador para volver a ejecutar el job:

- Aumentando el valor del atributo start-limit
- Ejecutar el job como un nuevo JobInstance

Volver a ejecutar un Step

- Cuando nuestro job es restartable, si el job no finalizó correctamente y volvemos a lanzar el job; no ejecutará los steps cuyo estado sea COMPLETED.
- Si queremos que un Step completed se vuelva a ejecutar al reiniciar un job, definiremos el atributo **allow-start-if-complete=”true”** en el tasklet.

Por defecto **allow-start-if-complete=”false”**

Saltando errores de lectura

- Hay ocasiones en las que no queremos que un **Step** resulte fallido porque se han producido errores durante el procesamiento.

  Ejemplos:

  - Carga de un catálogo de productos.
  - Carga de la lista de proveedores.
- Podemos indicar el número máximo de errores permitidos y clases de las excepciones que representan el error

Saltando errores lectura

```
<batch:step id="step1">
   <batch:tasklet>
      <batch:chunk reader="flatFileItemReader" writer="itemWriter"
            commit-interval="10" skip-limit="10">
         <skippable-exception-classes>
            <include class="org.springframework.batch.item.file.
   FlatFileParseException"/>
         </skippable-exception-classes>
      </batch:chunk>
   </batch:tasklet>
</batch:step>
```

- El valor de **skip-limit** es la suma entre errores de lectura/procesamiento/escritura del *chunk*.

- Podemos incluir clases de excepciones con el tag **include** y excluir con el tag **exclude**

Reintento lectura/escritura

- En ocasiones nos interesa reintentar la ejecución de un chunk porque se ha producido una excepción que en el siguiente intento podría no producirse.

  Ejemplos:

  - Recurso no está disponible (error al acceder al recurso por ftp/http)
  - DeadLock
- Podemos indicar el número de reintentos y las clases de excepciones que representan el error que hay que reintentar.

Reintentos de lectura/escritura

```
<batch:step id="step1">
   <batch:tasklet>
      <batch:chunk reader="itemReader" writer="itemWriter"
             commit-interval="2" retry-limit="3">
         <retryable-exception-classes>
            <include class="org.springframework.dao.
   DeadlockLoserDataAccessException"/>
         <s/retryable-exception-classes>
      </batch:chunk>
   </batch:tasklet>
</batch:step>
```

- El valor de **retry-limit** es el número de reintentos.

- Podemos incluir clases de excepciones con el tag **include** y excluir con el tag **exclude**

Controlando Rollback

- Cualquier excepción lanzada por un **ItemWriter** causa un rollback en la transacción controlada por el Step.
- Si se configura *skip*, las excepciones lanzadas por el ItemReader no causarán rollback.
- Sin embargo en ocasiones nos interesa que las excepciones lanzadas por el *ItemWriter no causen rollback* porque no se ha producido una acción que invalide la transacción.

Controlando Rollback

```
<batch:step id="step1">
   <batch:tasklet>
      <batch:chunk reader="itemReader" writer="itemWriter" commit-
   interval="2"/>
      <batch:no-rollback-exception-classes>
         <include class="org.springframework.batch.item.validator.
   ValidationException"/>
      </batch:no-rollback-exception-classes>
   </batch:tasklet>
</batch:step>
```

STEP Listeners

ItemReader transaccional

- El ItemReader se consume de forma secuencial. El Step almacena los datos leídos; de forma que en caso de rollback no se necesite volver a leer los items.
- Sin embargo, cuando el reader consume recursos transaccionales, como una cola JMS, como la cola está asociada a la transacción que ha sufrido el rollback, se restauran los mensajes consumidos de la cola.
- Se puede configurar un step para que no almacen los items leídos, usando **is-reader-transactional-queue="true"** en el chunk.

Listeners

- Existen eventos durante la ejecución de un Step en los que usuario puede intervenir.
- Se realiza mediante la configuración de listeners en el **Step**, **Tasklet** o **Chunk**.
- Los ItemReader, ItemProcessor o ItemWritter que implementen alguna de las interfaces; serán configurados automáticamente.

Ejemplos:

- Generar un registro de sumario al final del procesamiento de un fichero.

Listeners: Interfaces

- **StepExecutionListener**
  - Notificación antes de empezar la ejecución y una vez finalizada, tanto si ha finalizado correctamente como si lo ha hecho con error.
- **ChunkListener**
  - Notificación antes de que el chunk empiece el procesamiento o una vez lo haya completado.
- **ItemReadListener**
  - Notificación antes de leer un item, después de leer o cuando se ha producido un error. Este caso permite registrar el error en el log.

Listeners: Interfaces

- **ItemProcessListener**
  - Notificación antes de procesar un item, después de procesarlo o si se ha producido un error.
- **ItemWriteListener**
  - Notificación antes de escribir un item, después de procesarlo o si se ha producido un error.
- **SkipListener**
  - Seguimiento de los elementos que se han saltado (en la lectura/procesamiento/escritura)

Listeners

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

STEPs sin procesamiento CHUNK

Step : Otras formas de procesamiento

- El procesamiento por conjuntos (chunk) no es la única forma de procesar un Step.

Ejemplos:

- Descargar un fichero de un servidor FTP
- Invocar un STORED PROCEDURE
- Llamar a un script

Step: Otras formas de procesamiento

- **TaskletStep**: Es un step que ejecuta un tasklet
- **Tasklet** es una interface con un único método *execute* que se ejecutará de forma repetida hasta que devuelva RepeatStatus.FINISHED o lance una excepción como señal de error.
- Cada llamada un tasklet se envuelve en una transacción

Step: Otras formas de procesamiento

- El ejemplo del job HelloWorld, que vimos es un ejemplo.

```
 <bean id="helloTask" class="com.batch.springbatch.jobs.tasks.HelloWorldTask">
</bean>


<batch:job id="helloWorldJob">
    <batch:step id="step1">
       <batch:tasklet ref="helloTask"/>
    </batch:step>
</batch:job>
```

- El atributo ref del tasklet, le indica a SpringBatch que ha de crear un TaskletStep para procesar el step.

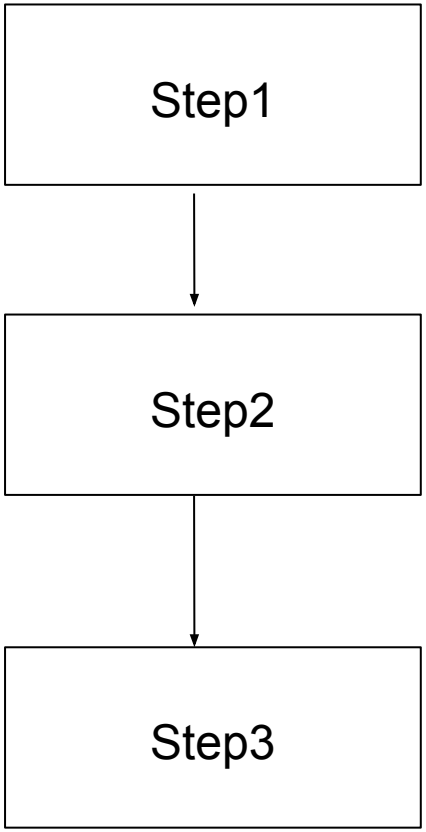
Control de Flujo

Control de flujo de Steps

- Flujo secuencial
- Flujo condicional
- Finalización del Job
- Programar las decisiones de flujo
- Flujos paralelos

Secuencial

- El escenario más simple es un job donde todos sus Steps se ejecutan de forma secuencial.




```
<batch:job id="job">
    <batch:step id="step1" next="step2" />
    <batch:step id="step2" next="step3"/>
    <batch:step id="step3"  />
</batch:job>
```

- Step1 será el primer step en ejecutarse (porque es el primero definido en la lista de steps del job).
- Si se ejecuta correctamente, se pasará a ejecutar el Step2.
- Sino, se el job terminará y no se ejecutarán el resto de steps.

BatchStatus: El estado de JobExecution y StepExecution

- **BatchStatus** es un enumeration usado como tipo de datos de un atributo de **JobExecution** y **StepExecution** y es informado por SpringBatch para registrar el estado de un job o un step. Sus posibles valores son:

  - COMPLETED
  - FAILED
  - STARTING
  - STARTING
  - STOPPING
  - STOPPED
  - ABANDONED
  - UNKNOWN

Gestionando la transición: **ExitStatus**

- El valor de ExitStatus es el que se utiliza al establecer los condicionales del job para trazar el control de flujo.
- **ExitStatus** representa el estado de un Step cuando ha finalizado su ejecución. Sus valores son FAILED o COMPLETED.
- Sin embargo podemos definir valores diferentes, definiendo **StepExecutionListener** devolviendo el valor que nos interese en el método ExitStatus afterStep(StepExecution se).

Gestionando la transición: **Elementos de transición**

- Para gestionar escenarios más complejos, se pueden definir **elementos de transición** dentro de la definición de un Step.
- Elementos de transición
  - next El siguiente Step a ejecutar
  - end *El Job finaliza con batchStatus COMPLETED*
  - fail *El Job finaliza con batchStatus FAILED*
- Los elementos de transición indican la acción a llevar a cabo en función del **ExitStatus** del Step

Gestionando la transición: **Elementos de transición**

```
<batch:step id="step2">
    <end on="NO CONNECTION"/>
    <fail on="FAILED" exit-code="EARLY TERMINATION"/>
    <next on="*" to="step3"/>
</batch:step>
```

- El Job finaliza con batchStatus COMPLETED si exitStatus del Step es “NO CONNECTION”
- El Job finaliza con batchStatus FAILED y exitStatus EARLY TERMINATION si exitStatus del Step es “FAILED”
- El Job sigue pasa a ejecutar step3; si exitStatus del Step es cualquier otro valor

Gestionando la transición: **Finalización del JOB**

- Si no se definen transiciones para un Step; el Job terminará y su estado será:
  - Si step.ExitStatus del Step == FAILED

  job.batchStatus = FAILED

  job.exitStatus = FAILED
  - Sino

  job.batchStatus = COMPLETED

  job.exitStatus = COMPLETED

Gestionando la transición: **Programando las decisiones**

- En ocasiones es necesaria más información para decidir el flujo de ejecución de un Job.
- La interfaz **JobExecutionDecider** y el elemento **decision** dentro de un Step.

Gestionando la transición: **Programando las decisiones**

```
<batch:job id="job">
    <batch:step id="step1" next="decision" />
    <batch:decision id="decision" decider="decider">
        <next on="FAILED" to="step2" />
        <next on="COMPLETED" to="step3" />
    </batch:decision>
    <batch:step id="step2" next="step3"/>
    <batch:step id="step3" />
</batch:job>
```

- Cuando finaliza **step1**, el elemento de decisión controlará el flujo hacia el **step2** o al **step3**.

- Se ha de crear un bean que implemente la interfaz JobExecutionDecider y en el método **decide**(JobExecution jobExecution, StepExecution stepExecution) devolver el valor que se evaluará.

Procesos en paralelo

- Se puede ejecutar procesos en paralelo se elemento **split** que contiene un elemento **flow** por cada uno de procesos que se pueden realizar en paralelo.
- Puede configurarse un **task-executor** en el elemento split para definir qué implementación concreta se utilizará para ejecutar cada **flow**.

Por defecto se usa SyncTaskExecutor, pero se necesita un executor asíncrono para ejecutarlos en paralelo.

Procesos en paralelo

```
<batch:job id="job">
  <batch:split id="split1" next="step4" task-executor="tExecutor">
    <batch:flow>
        <batch:step id="step1" next="step2"/>
        <batch:step id="step2"/>
    </batch:flow>
    <batch:flow>
        <batch:step id="step3" parent="s3"/>
    </batch:flow>
   </batch:split>
   <batch:step id="step4" parent="s4"/>
</batch:job>

<bean id="tExecutor" class="org.springframework.core.task.
   SimpleAsyncTaskExecutor"/>
```

Ejecutando comandos de Sistema

- Muchos Jobs requieren que se llame a un comando externo => **org.springframework.batch.core.step.tasklet.SystemCommandTasket**

```
<bean class="org.springframework.batch.core.step.
tasklet.SystemCommandTasklet">
    <property name="command" value="echo hello" />
</bean>
```

Propiedades extras:

- **workingDirectory**:
- **environmentParams**
- **timeout** : Tiempo máximo de ejecución en milisegundos
- **interruptOnCancel**: Si queremos que el tasklet interrumpa el hilo si se supera el timeout o si se cancela el job
- **taskExecutor**: TaskExecutor que ejecutará el proceso

Pasar información de un Step a otro

- ExecutionContext
  - **Contexto de ejecución de Job**: El tiempo de vida es durante la ejecución del Job.

  Se actualiza cada vez que finaliza un **Step**
  - **Contexto de ejecución de Step**: El tiempo de vida es durante la ejecución del Step.

  Se actualiza cada vez que finaliza un **Chunk**

Pasar información de un Step a otro

- Los datos se han de almacenar en el **contexto de ejecución del Step** mientras el step está en ejecución y se ha de **promocionar al contexto de ejecución del Job** cuando el Step haya finalizado.

Almacenar información en el Contexto de Ejecución

Leer del contexto de Ejecución

```
… (ExecutionContext context) {

context.get(“variable”);

}
```

Escribir en el contexto de Ejecución

```
… (ExecutionContext context) {

Context.set(“variable”, valor);

}
```

Promocionando contexto Step

Step que genera la información

```
public class MyWriter implements ItemWriter, StepExecutionListener {
  StepExecution context = null;

  public void write(List list) {
    context.put(“key”,value);
  }

  public void beforeStep(StepExecution stepExecution)
     context = stepExecution;
   }
  public void afterStep(StepExecution stepExecution)
   }
}
```

```
<step id="mystep">
  <tasklet>
  </tasklet>
  <listeners>
    <listener ref="myPromotionListener"/>
  </listeners>
</step>
```

Promocionando contexto Step

Step que procesa la información

```
<bean id="myPromotionListener" class=
"ExecutionContextPromotionListener">
  <property name="keys" value="key"/>
</bean>
```

```
public class MyOtherWriter implements ItemWriter, StepExecutionListener {
  String value = null;

    public void beforeStep(StepExecution stepExecution)
      JobExecution jobExecution = stepExecution.getJobExecution();
      ExecutionContext jobContext = jobExecution.getExecutionContext();
      value= jobContext.get("key");
   }
  public void afterStep(StepExecution stepExecution)
   }
}
```

Promocionando contexto Step

Step que procesa la información

```
<bean id="myPromotionListener" class=
"ExecutionContextPromotionListener">
  <property name="keys" value="key"/>
</bean>
```

```
public class MyOtherWriter implements ItemWriter, StepExecutionListener {
  String value = null;

    public void beforeStep(StepExecution stepExecution)
      JobExecution jobExecution = stepExecution.getJobExecution();
      ExecutionContext jobContext = jobExecution.getExecutionContext();
      value= jobContext.get("key");
   }
  public void afterStep(StepExecution stepExecution)
   }
}
```

Concurrencia

Concurrencia
Step multihilo

Usar TaskExecutor (org.springframework.core.task.TaskExecutor) en un Step

Se utiliza la propiedad task-executor del tasklet y se define el *bean* que se encargará de ejecutarla

```
<step id="myStep">
    <tasklet task-executor="myTaskExecutor">
    </tasklet>
</step>

<bean id="myTaskExecutor" class="org.springframework.core.task.SimpleAsyncTaskExecutor">
</bean>
```

Existen diferentes implementaciones de TaskExecutor en Spring. Una de ellas es SimpleAsyncTaskExecutor

¿Qué sucede?

Cada **chunk** se ejecuta en su propio hilo de ejecución

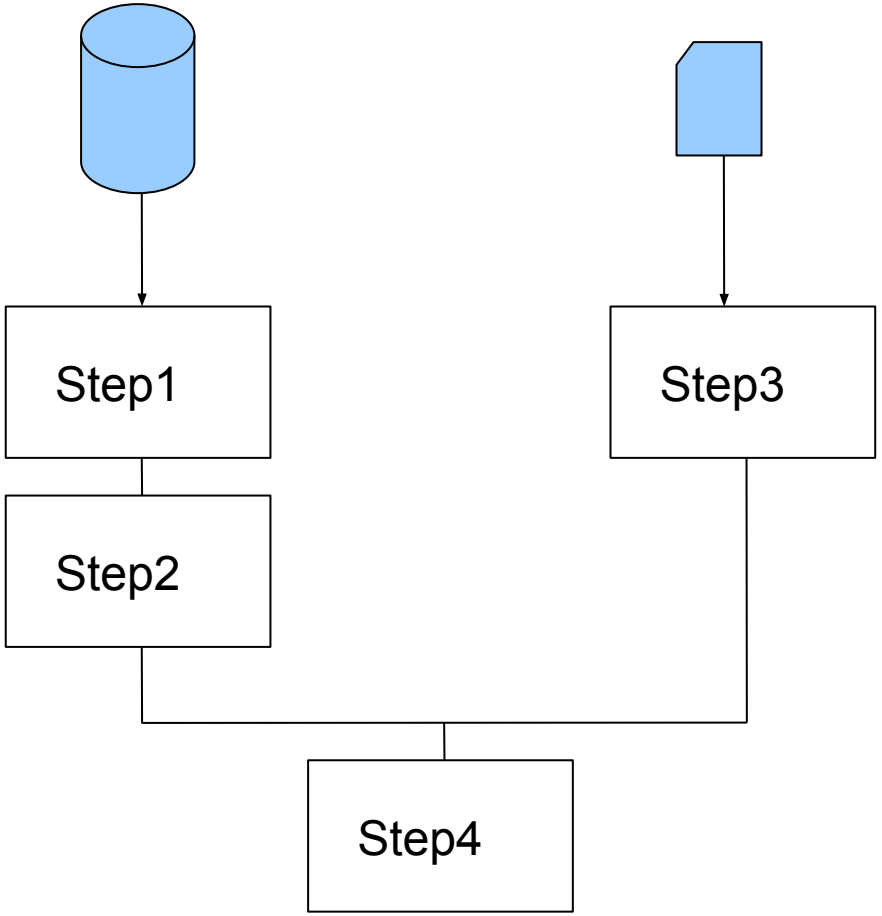
Concurrencia
Step multihilo

Aspectos a tener en cuenta:

- El ItemReader, ItemProcessor e ItemWriter deben ser Thread-Safe
- Los elementos no se procesarán en orden secuencial

Concurrencia
Steps paralelos

Establecer qué steps se pueden ejecutar en pararelo (en el mismo proceso)



El proceso que se realiza en los steps 1 y 2 (obtención de datos de la BBDD) se puede realizar en paralelo al proceso del Step3 (obtención de datos de un fichero). De forma que el proceso 4 se ejecuta una vez ambos han terminado

Muchas Gracias!!!!!