Spring Batch

Víctor Herrero Cazurro

Temario

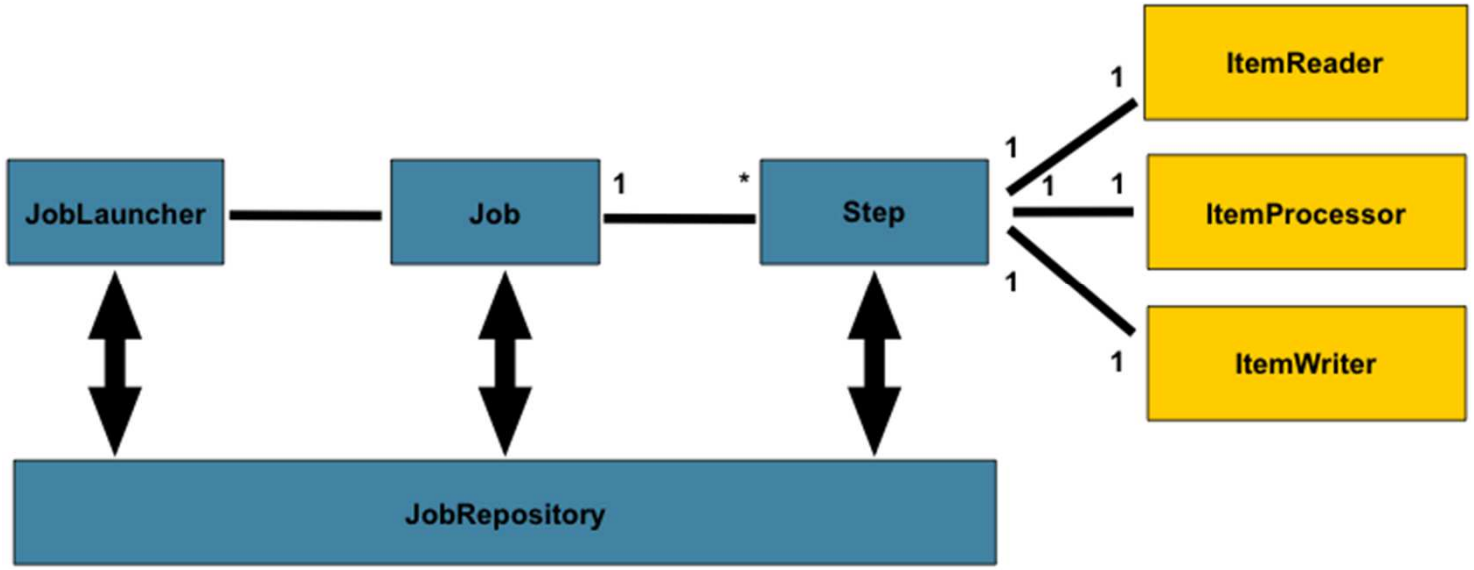
1. Introducción
2. Chunk-oriented-procesed
3. Configuración de entorno
4. JobRepository
5. JobLauncher
6. Job
7. Step
8. Tasklet
9. TaskletAdapter
10. Tasklet Chunk
11. Flujos
12. Listeners

Procesos batch

- Los procesos batch (o procesos por lotes) son aquellos programas que se ejecutan de seguido, sin interacción con usuarios.
- Grandes procesos, que mueven grandes cantidades de información.
- Se programa su ejecución en horarios con poco uso del entorno de ejecución (noches o fines de semana).

¿Que es Spring Batch?

- Framework basado en Spring para la definición de procesos por lotes.
- Especifica como se han de diseñar los procesos.
  - Estructura del proyecto

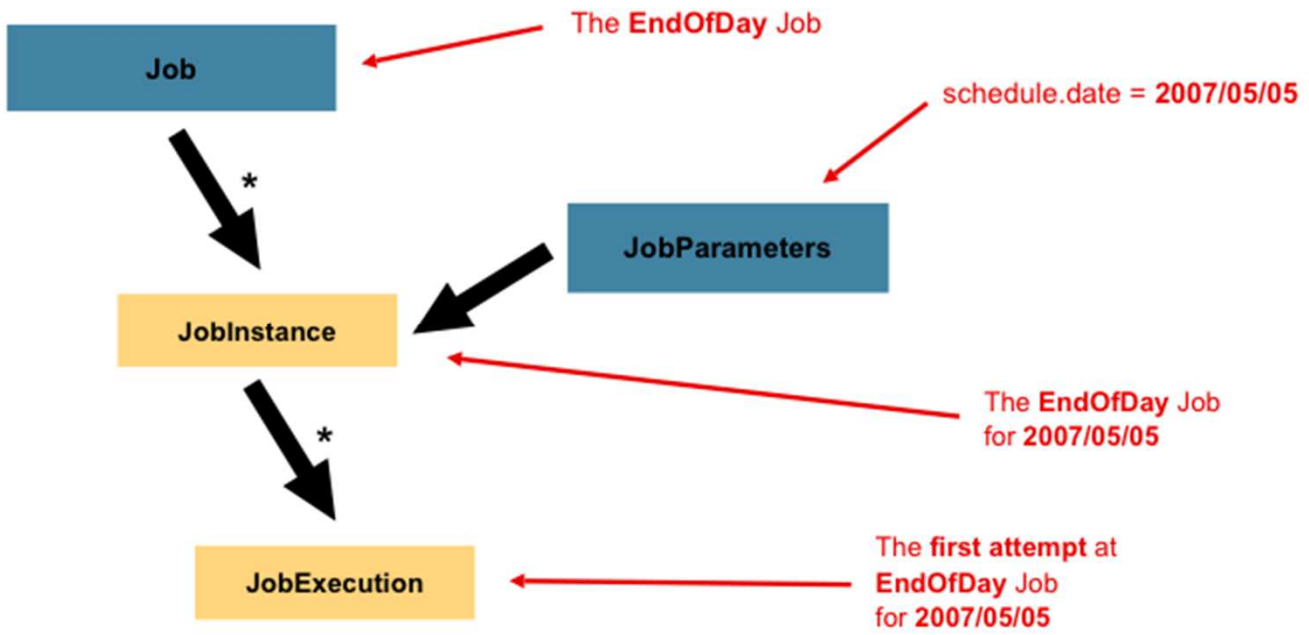


Job

- **Job**: Componente que representa el proceso.
- **JobInstance**: Cada una de las ejecuciones que se planean de un **Job**.
- **JobParameter**: Los parámetros que le llegan a un **JobInstance** y que los diferencia de otros **JobInstance** del mismo **Job**.
- **JobExecution**: El estado de la ejecución de un **JobInstance**.

Job

- Diagrama de relación de **Job**



JobExecution

- Los parámetros del **JobExecution** son

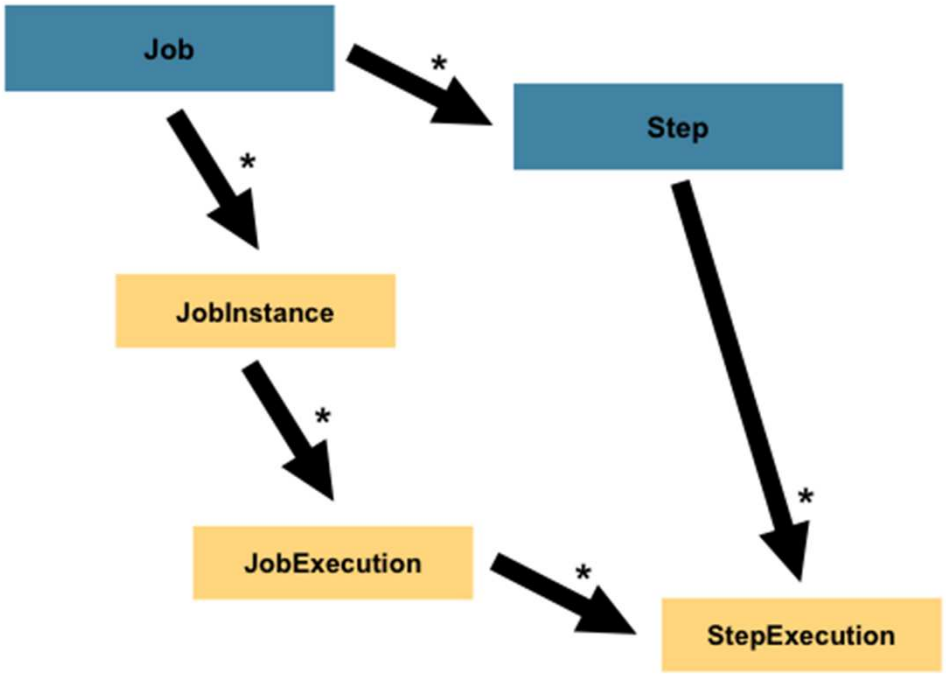
status	Objeto BatchStatus que indica el estado de la ejecución. Puede ser BatchStatus.STARTED, BatchStatus.FAILED o BatchStatus.COMPLETED
startTime	Objeto java.util.Date que indica cuando se arranca.
endTime	Objeto java.util.Date que representa cuando la Job termina, haya sido exitosa o no.
exitStatus	Estado de la Job al terminar.
createTime	Objeto java.util.Date que representa la primera vez que se persistió el Job, cuando todavía no se ha iniciado.
lastUpdated	Objeto java.util.Date reresenta la ultima vez que el Job ha sido persistido.
executionContext	Contiene toda la información necesaria entre persistencias en la ejecución
failureExceptions	Listado de Excepciones ocurridas a lo largo de la ejecución del Job.

Step

- **Step**: Cada paso que divide un **Job**. Un **Job** debe tener, al menos, un **Step**.
- **SetExecution**: Cada intento por ejecutar un **Step**. Es similar al **JobExecution**.

Step

- Diagrama de relación de **Step**.



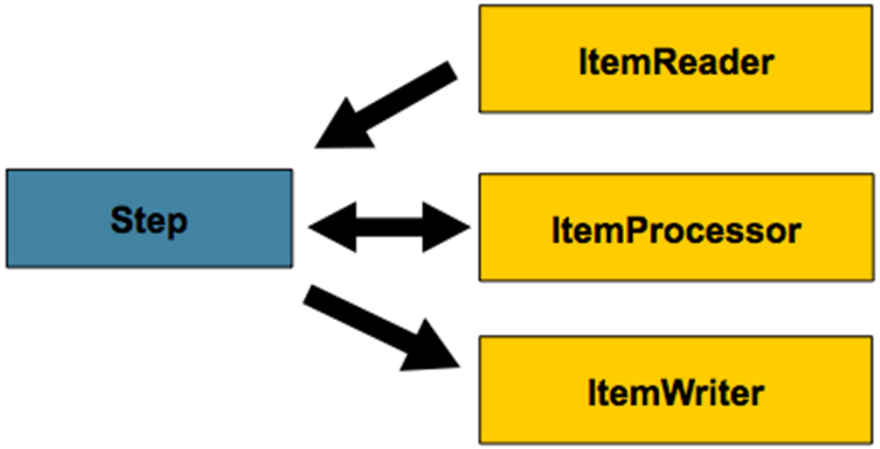
StepExecution

- Los parámetros del **StepExecution** son

Status	Objeto BatchStatus que indica el estado de la ejecución. Puede ser BatchStatus.STARTED, BatchStatus.FAILED o BatchStatus.COMPLETED
startTime	Objeto java.util.DateQue representa cuando se inicia la ejecución.
endTime	Objeto java.util.Date incida cuando termina la ejecución, haya ido bien o mal.l.
exitStatus	Indica el estado en el que finaliza la ejecución.
executionContext	Contiene toda la información necesaria entre persistencias en la ejecución
readCount	Numero de items leidos crrectamente.
writeCount	Numero de Items escritos correctamente
commitCount	Numero de Commits.
rollbackCount	Numero de Rollbacks.
readSkipCount	Numero de lecturas fallidas que han provocado el salto del item.
processSkipCount	Numero de procesamientos fallidos que han provado el salt de un item.
filterCount	Numero de Items tratados por el ItemProcesor.
writeSkipCount	Numero de escrituras fallidas que han provocado el salto de un item.

Step

- Los **Step** están compuestos por un **Tasklet** simple o por un **Chunk** con **ItemReader**, **ItemProccessor** e **ItemWriter**.



Step

- **ItemReader:** Componente dentro del **Step** que permite a este leer datos de un origen.
- **ItemProcessor:** Componente dentro del **Step**, que procesa lo obtenido por el **reader**.
- **ItemWriter:** Componente dentro del **Step**, que permite a este escribir datos en un destino. Si hay un **reader** debe haber un **writer**.

Conceptos

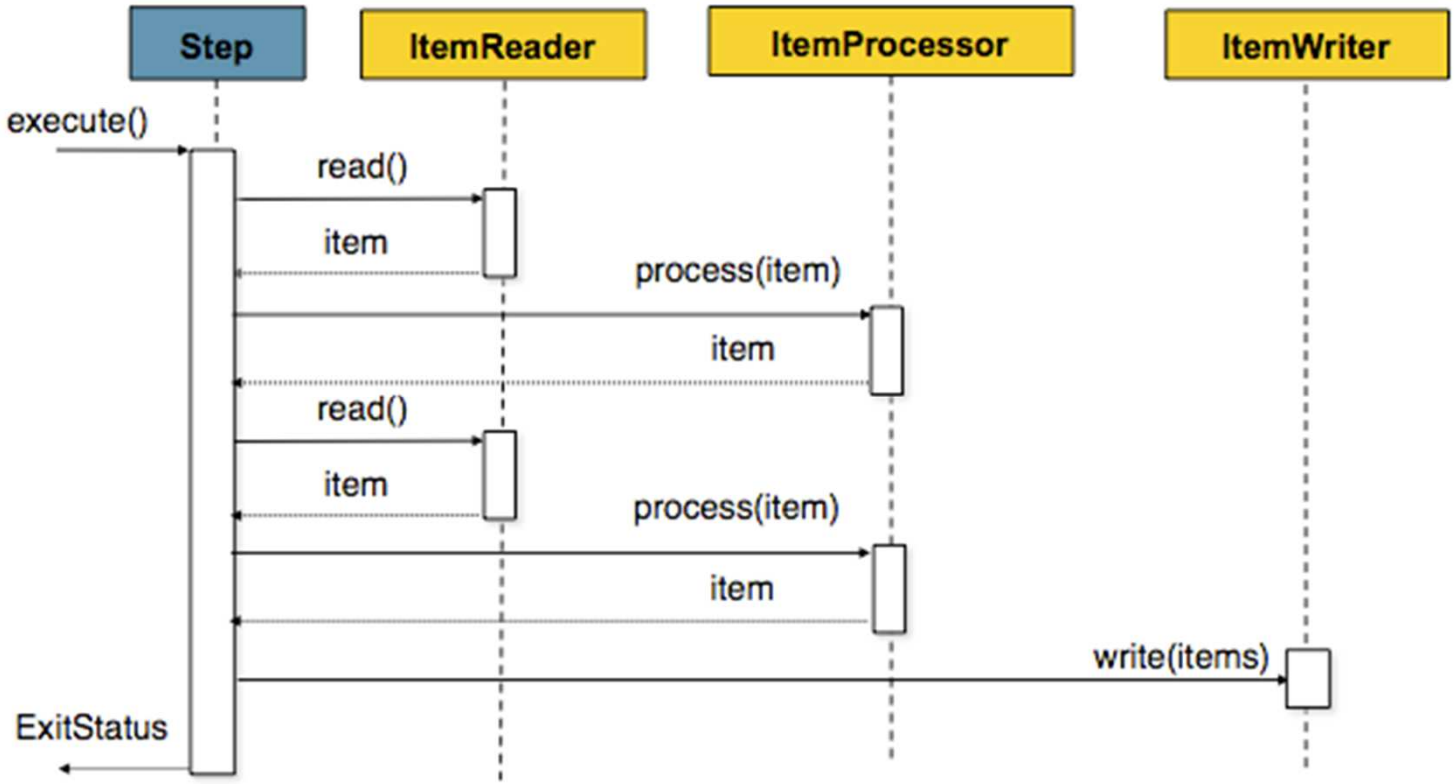
- **JobRepository**: Componente que mantiene la información de los procesos, como su numero o su estado.
- **JobLauncher**: Componente que lanza los procesos.

Chunk-Oriented Processing

- Estrategia muy empleada en la definición de Job.
- Indica que la **lectura** y el **procesamiento**, se realizan a trozos (**chunk**) y finalmente el resultado de todos los trozos se acaba **escribiendo**, todo ello en un entorno transaccional.
- El numero de **chunk** creados, se define con el intervalo de commit (**commit-interval**).

Chunk-Oriented Processing

- Esquema



Configuración de entorno

- Para poder trabajar con Spring Batch, se necesitan las dependencias.
- Con Maven

```
<dependency>
        <groupId>org.springframework.batch</groupId>
        <artifactId>spring-batch-core</artifactId>
        <version>2.2.5.RELEASE</version>
</dependency>
```

Configuración de entorno

- Si se desea una versión mas moderna de Spring Context y/o Transaction

```
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>3.2.8.RELEASE</version>
</dependency>
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-tx</artifactId>
        <version>3.2.8.RELEASE</version>
</dependency>
```

Configuración de entorno

- Si se va emplear persistencia, se necesitará también JDBC

```
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>3.2.8.RELEASE</version>
</dependency>
```

Configuración de entorno

- Si se van a procesar XML se puede emplear OXM.

```
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-oxm</artifactId>
        <version>3.2.8.RELEASE</version>
</dependency>
```

JobRepository

- Es el encargado de persistir los objetos **JobExecution** y **StepExecution**.
- Si se emplea la etiqueta del Namespace, por defecto se crea con las siguientes opciones.

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

- La implementación por defecto es JobRepositoryFactoryBean, que trabaja con BD.

JobRepository

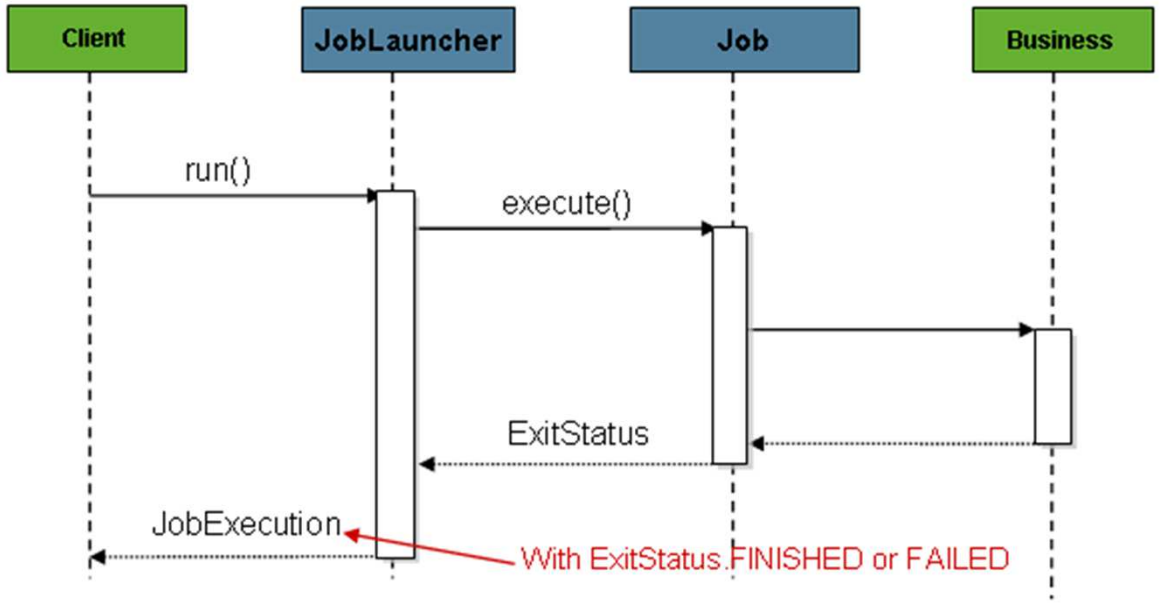
- Se puede definir una implementación en memoria.

```
<bean id="jobRepository"
class="o.s.b.core.repository.support.MapJobRepositoryFactoryBean">
        <property name="transactionManager"
                                        ref="transactionManager"/>
</bean>
```

JobLauncher

- El mas básico sería **SimpleJobLauncher**.

```
<bean id="jobLauncher"
                    class="o.s.b.core.launch.support.SimpleJobLauncher">
        <property name="jobRepository" ref="jobRepository" />
</bean>
```



JobLauncher

- En un entorno web, dado que la petición requiere una respuesta mas o menos inmediata y el job, tendra una duración bastante elevada, es preciso indicar que la ejecución sea asincrona, para que la respuesta pueda ser generada aunque no haya terminado el Job.

```
<bean id="jobLauncher"
                    class="o.s.b.core.launch.support.SimpleJobLauncher">
        <property name="jobRepository" ref="jobRepository" />
        <property name="taskExecutor">
                    <bean class="o.s.c.task.SimpleAsyncTaskExecutor" />
        </property>
</bean>
```

JobLauncher

- Ejemplo de lanzamiento de **Job** desde un controlador de Spring MVC

```
@Controller
public class JobLauncherController {

		@Autowired
		JobLauncher jobLauncher;

		@Autowired
		Job job;

		@RequestMapping("/jobLauncher.html")
		public void handle() throws Exception{
			jobLauncher.run(job, new JobParameters());
		}
}
```

JobLauncher

- Ejemplo de arranque de un Job con Spring Test

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {
"classpath:/curso/spring/batch/ej01/spring-batch-config-namespace.xml"})
public class ImprimirHolaMundoTest {
        @Autowired private SimpleJobLauncher launcher;
        @Autowired @Qualifier("trabajoBatch") private Job job;
        @Test public void iniciarJob() throws Exception {
                JobParametersBuilder builder =
                                new JobParametersBuilder();
                builder.addDate("Ejecucion", new Date());
                builder.addString("jobName",
                                "Imprimir hola mundo por consola");
                JobParameters parameters = builder.toJobParameters();
                launcher.run(job, parameters);
        }
}
```

Configurar un Job

- Los Job necesitan la siguiente información:
  - **Name**.
  - **JobRepository**. Sino se define se busca un bean llamado **jobRepository**.
  - Lista de **Step**.

Configurar un Job

- Ejemplo de creación de un **Job**

```
<batch:job id="trabajoBatch">
        <batch:step id="primerPaso" next="segundoPaso">
                <batch:tasklet ref="imprimirHola“/>
        </batch:step>
        <batch:step id="segundoPaso" next="tercerPaso">
                <batch:tasklet ref="imprimirMundo“/>
        </batch:step>
        <batch:step id="tercerPaso">
                <batch:tasklet ref="imprimirExclamacion“/>
        </batch:step>
</batch:job>
```

Configurar un Step

- Un **Step** podrá estar formado por
  - **tasklet**: Tarea simple o Chunk.
  - **end**: Permite indicar con que **ExitStatus** se pone el **BatchStatus** a COMPLETED.
  - **stop**: Permite indicar con que **ExitStatus** se pone el **BatchStatus** a STOPPED.
  - **next**: Permite indicar cual es el siguiente Step (**to**) dependiendo de que ExitStatus (**on**).
  - **fail**: Permite indicar con que **ExitStatus** se pone el **BatchStatus** a FAILED.
  - **listeners**: Colección de **Listeners** a aplicar.

Herencia

- Se puede establecer herencia entre Step, para reutilizar configuraciones

```
<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>
```

Herencia

- Los Step se pueden definir como abstractos para que no sea utilizados directamente.

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

Re-arranque

- Es posible que se quiera controlar si una tarea se puede o cuando puede re-arrancarse.
- El re-arranque permite que procesos que han fallado se vuelvan a ejecutar.
- Se pueden definir parámetros de <job> como
  - **restartable**: Se puede re-arrancar.
- Se pueden definir parámetros de <tasklet> como
  - **start-limit**: Numero de veces a re-arrancar como máximo.
  - **allow-start-if-complete**: Solo se puede re-arrancar si termino con éxito.

Registros no procesados

- Es posible que en el procesado de registros, existan fallos, y no siempre estos fallos han de ocasionar el parón del Job.
- Para indicar que un fallo no ocasiona el paro del Job.

```
<step id="step1">
  <tasklet>
    <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>
    </chunk>
  </tasklet>
</step>
```

Registros no procesados

- Se puede indicar el numero de registros máximos que se pueden no procesar con la propiedad de <**chunk**> **skip-limit**.

Reintentos por bloqueos

- Es posible que en el procesado de registros, existan fallos, por que exista un bloqueo temporal sobre algún recurso necesario, se puede indicar que se reintente el procesado.

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

Rollback

- En general cualquier excepción que se lance desde el procesado, provocará un **rollback**, si se desean excluir algún tipo de ellas de esa casuística

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

Tasklet

- Clases que implementan la interface

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

- Serán incluidas dentro de los **Step**, con la etiqueta <**batch:tasklet**>.

```
public class ImprimirTasklet implements Tasklet {
        private String mensaje;
        public String getMensaje() {return mensaje; }
        public void setMensaje(String mensaje) {this.mensaje = mensaje;}
        @Override
        public RepeatStatus execute(StepContribution contribution,
                                    ChunkContext chunkContext) throws Exception {
                System.out.print(mensaje);
                return RepeatStatus.FINISHED;
        }
}
```

Tasklet

- Deben de implementar el método **execute**, que representa la ejecución del **Step**.
- Recibe
  - **StepContribution**: Representa el estado del **Step**, pudiendo ser editado por el **Tasklet**.
  - **ChunkContext**: Encapsula el **StepContext**.

TaskletAdapter

- Clase especial, que permite invocar un método de una clase ya definida, sin necesidad de crear la clase Tasklet.

```
<bean id="myTasklet" class="o.s.b.core.step.tasklet.MethodInvokingTaskletAdapter">
        <property name="targetObject">
                <bean class="org.mycompany.FooDao"/>
        </property>
        <property name="targetMethod" value="updateFoo" />
</bean>
```

Tasklet Chunk

- Al definir un Step Chunk, se han de definir
  - **ItemReader**.
  - **ItemWriter**.
  - **TransactionManager**.
  - **JobRepository**. Repositorio de Job, donde se guardaran periódicamente **StepExecution** y **ExecutionContext**.
  - **Commit-interval**. Items procesados para hacer el commit.

Tasklet Chunk

- Mínima configuración para un Tasklet Chunk

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

TransactionManager

- Es el gestor transaccional que controla la transacción que se crea entre los distintos Items ejecutados en el Chunk (Reader, Procesor, Writer).
- Deberá ser de tipo **AbstractPlatformTransactionManager**.
- Los distintos módulos de Spring proporcionan distintas implementaciones.

ItemReader

- Permite la lectura de información de orígenes de datos.
- Existen varias implementaciones ofrecidas por el API.
  - **FlatFile**: Lee líneas de un fichero de texto plano.
  - **Xml**: Lee nodos de un xml.
  - **Database**: Lee registros de un base de datos.
- Todas implementan la interface **ItemReader**.

FlatFileItemReader

- Permite leer de un fichero de texto plano y volcarlo en un objeto.
- Necesita que se le defina un **dataMapper** que realice el marshall.

```
<bean id="processFileItemReader"
                        class="org.springframework.batch.item.file.FlatFileItemReader">
        <property name="resource" value="${input.file}"/>
        <property name="lineMapper" ref="fieldSetMapper"/>
</bean>
```

BeanWrapperFieldSetMapper

- Un Mapper que transforma un texto en un Bean aportado como referencia.

```
<bean id="fieldSetMapper"
                        class="o.s.b.item.file.mapping.BeanWrapperFieldSetMapper">
        <property name="prototypeBeanName" value="player" />
</bean>

<bean id="player“  class="org.springframework.batch.sample.domain.Player"
        scope="prototype" />
```

- Siendo la clase Player

```
public class Player implements Serializable {
        private String ID;
        private String firstName;
        private String position;
        // setters and getters...
}
```

BeanWrapperFieldSetMapper

- Y los datos de entrada soportados

```
ID,firstName,position
"AbduKa00,Karim,rb",
"AbduRa00,Rabih,rb",
"AberWa00,Walter,rb",
"AbraDa00,Danny,wr",
"AdamBo00,Bob,te",
"AdamCh00,Charlie,wr"
```

ItemWriter

- Análogo al **ItemReader**, pero para escribir.
- Las implementaciones de la interface **ItemWriter**.

FlatFileItemWriter

- Permite escribir en un fichero de texto plano objetos.
- Necesita un **LineAggregator**, que se encarga de pasar un Objeto a String.
  - Una de las implementaciones disponibles seria **PassThroughLineAggregator**, que invoca **toString**().
- También se puede definir para algunos **LineAggregator** un **FieldExtractor**, que transforma el **Object** en un Array de Objetos

FlatFileItemWriter

- Un ejemplo de FlatItemWriter

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

FlatFileItemWriter

- En este aso se procesarán objetos de la tipología

```
public class CustomerCredit {
    private int id;
    private String name;
    private BigDecimal credit;
    //getters and setters removed for clarity
}
```

ItemProcessor

- Permiten incorporar una lógica de negocio al proceso de lectura/escritura.
- Implementaran la interface **ItemProcessor**.

```
public class FooProcessor implements ItemProcessor<Foo,Bar>{
        public Bar process(Foo foo) throws Exception {
                return new Bar(foo);
        }
}
```

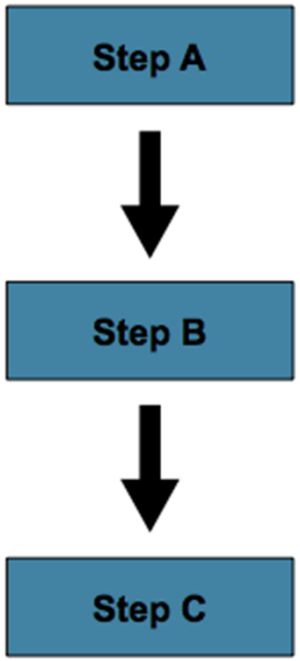
CompositeItemProcessor

- Si se desea unir varios **ItemProcessor** en un mismo **Step**, se puede componer un **CompositeItemProcessor**, que ira ejecutando cada uno de ellos en el orden definido.

```
<bean id="compositeItemProcessor"
        class="org.springframework.batch.item.support.CompositeItemProcessor">
        <property name="delegates">
                <list>
                        <bean class="..FooProcessor" />
                        <bean class="..BarProcessor" />
                </list>
        </property>
</bean>
```

Flujos secuenciales

- Se pueden definir flujos secuenciales en los Job, empelando la propiedad **next** de la etiqueta <**step**>



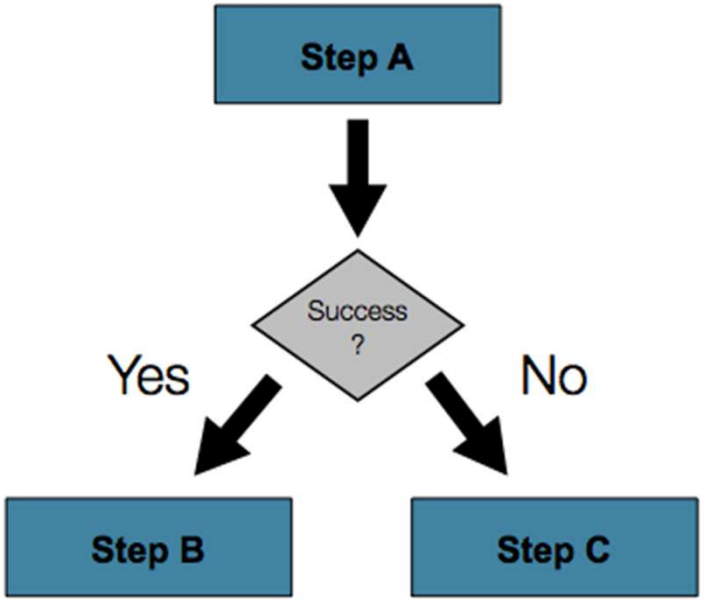
Flujos secuenciales

- Ejemplo de Flujo secuencial.

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

Flujos condicionales

- Se pueden definir flujos condicionales en los **Step**, empleando la etiqueta <**next**>, con la que se indica el siguiente **Step** basándose en el **ExitStatus** del actual.



Flujos condicionales

- Ejemplo de Flujo condicionales.

```
<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>
```

- En la propiedad **on** se puede emplear
  - * -> Cualquier numero de caracteres.
  - ? -> Cualquier carácter.

Flujos paralelos

- Hay un tipo de **Step** especial, que permite la ejecución en paralelo de bloques de **Step**.
- La etiqueta que lo representa es el <**Split**>

```
<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"/>
```

Flujos paralelos

- Los <**flow**> o grupos de Stop, se pueden re-factorizar para su reutilización

```
<job id="job">
        <flow id="job1.flow1" parent="flow1" next="step3"/>
        <step id="step3" parent="s3"/>
</job>

<flow id="flow1">
        <step id="step1" parent="s1" next="step2"/>
        <step id="step2" parent="s2"/>
</flow>
```

Listeners

- Permiten escuchar una serie de eventos que se pueden producir en la ejecución del **Step**.
- Para crear un listener se ha de declarar en el el **Step**.

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

Listeners

- Los Listener, pueden ser de distintos tipos, dependiendo de que evento escuchen.
  - StepExecutionListener
  - ChunkListener
  - ItemReadListener
  - ItemReadListener
  - ItemProcessListener
  - ItemWriteListener
  - SkipListener

@VictorHerrero1

Víctor Herrero Cazurro

victorherrerocazurro

victorherrerocazurro