



Formación

Curso Spring Batch

indra



Formador

Ana Isabel Vegas



INGENIERA INFORMÁTICA con Master Máster Universitario en Gestión y Análisis de Grandes Volúmenes de Datos: Big Data, tiene la certificación PCEP en Lenguaje de Programación Python y la certificación JSE en Javascript. Además de las certificaciones SCJP Sun Certified Programmer for the Java 2 Platform Standard Edition, SCWD Sun Certified Web Component Developer for J2EE 5, SCBCD Sun Certified Business Component Developer for J2EE 5, SCEA Sun Certified Enterprise Architect for J2EE 5.

Desarrolladora de Aplicaciones FULLSTACK, se dedica desde hace + de 20 años a la CONSULTORÍA y FORMACIÓN en tecnologías del área de DESARROLLO y PROGRAMACIÓN.



training@iconotc.com

❑ **Duración:** 8 horas

❑ **Modalidad:** On-line

❑ **Fechas/Horario:**

- Días 6, 7, 8 y 9 Octubre 2025
- Horario 15:30 – 17:30 hs.

❑ **Contenidos:**

- Arquitectura
- Artefactos esenciales
- Configuración de una tarea (Job)
- Configuración de un paso (step) de una tarea
- Mecanismos de lectura y escritura
- Escalabilidad y concurrencia
- Gestión de errores
- Testing

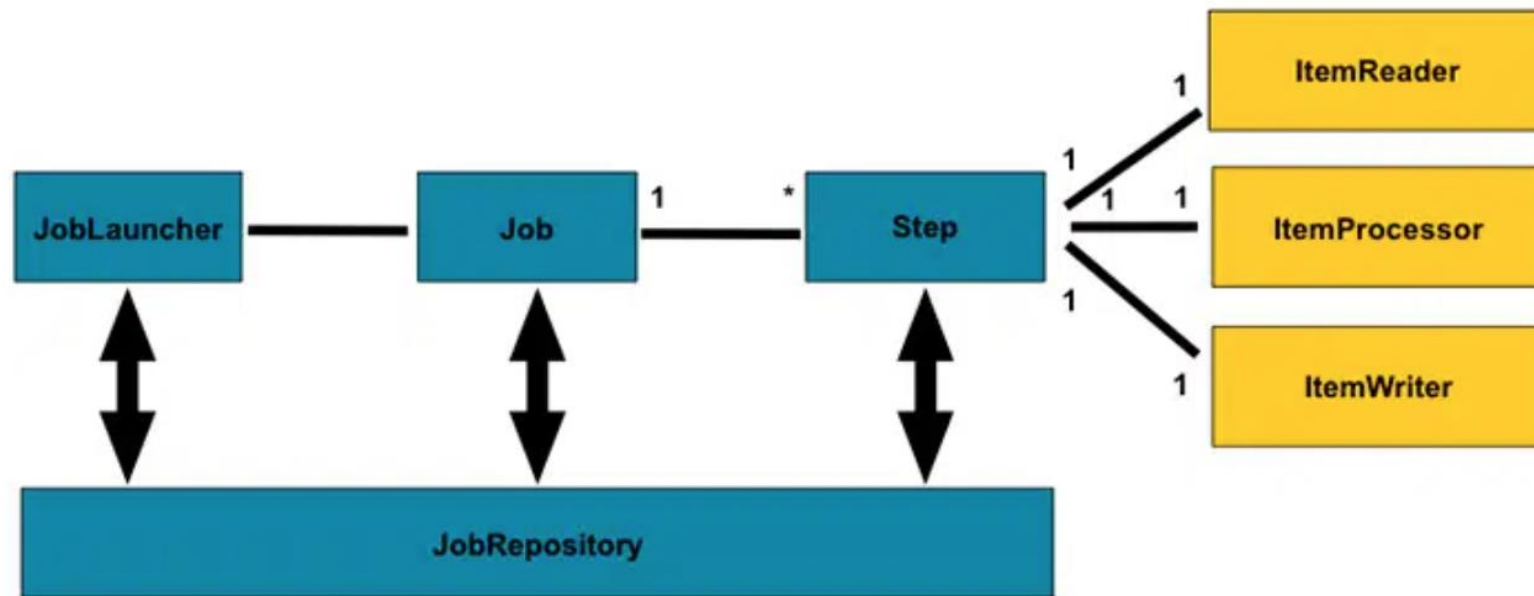
Arquitectura

Tema 1

Que es Spring Batch .-

- Spring Batch es un framework ligero enfocado específicamente en la creación de procesos batch.
- Además de marcar unas directrices para el diseño de procesos, Spring Batch proporciona con una gran cantidad de componentes que intentan dar soporte a las diferentes necesidades que suelen surgir a la hora de crear estos programas: trazas, transaccionalidad, estadísticas, paralelismo, particionamiento, lectura y escritura de datos, etc...
- Los procesos batch (o procesos por lotes) acostumbran a ser aquellos programas que se lanzan bajo una determinada planificación y por lo tanto no requieren ningún tipo de intervención humana. Suelen ser procesos relativamente pesados, que tratan una gran cantidad de información, por lo que normalmente se ejecutan en horarios con baja carga de trabajo para no influir en el entorno transaccional.

Componentes de Spring Batch .-



Componentes de alto nivel .-

- **JobRepository:** Se encarga de guardar los metadatos de los trabajos batch, como el estado de las ejecuciones, permitiendo su reinicio y la continuación desde donde se detuvieron.
- **JobLauncher:** Este componente es el responsable de iniciar un Job, recibiendo los parámetros de entrada necesarios para ejecutarlo.
- **Job:** Representa un proceso por lotes completo y se define como un contenedor de uno o más Step.
- **Step:** Es una unidad lógica de un Job y representa una fase específica del procesamiento. Un Job debe tener al menos un Step.

Componentes de procesamiento de datos .-

- Estos componentes forman la base del "patrón de tres pasos" (reader, processor, writer) y se encuentran dentro de cada Step.
 - **ItemReader**: Lee elementos de una fuente de datos, como una base de datos o un archivo.
 - **ItemProcessor**: (Opcional) Procesa los elementos leídos por el ItemReader, realizando transformaciones, cálculos o validaciones.
 - **ItemWriter**: Escribe o guarda los elementos, ya sea los leídos directamente por el ItemReader o los procesados por el ItemProcessor, en un destino deseado.

Artefactos esenciales

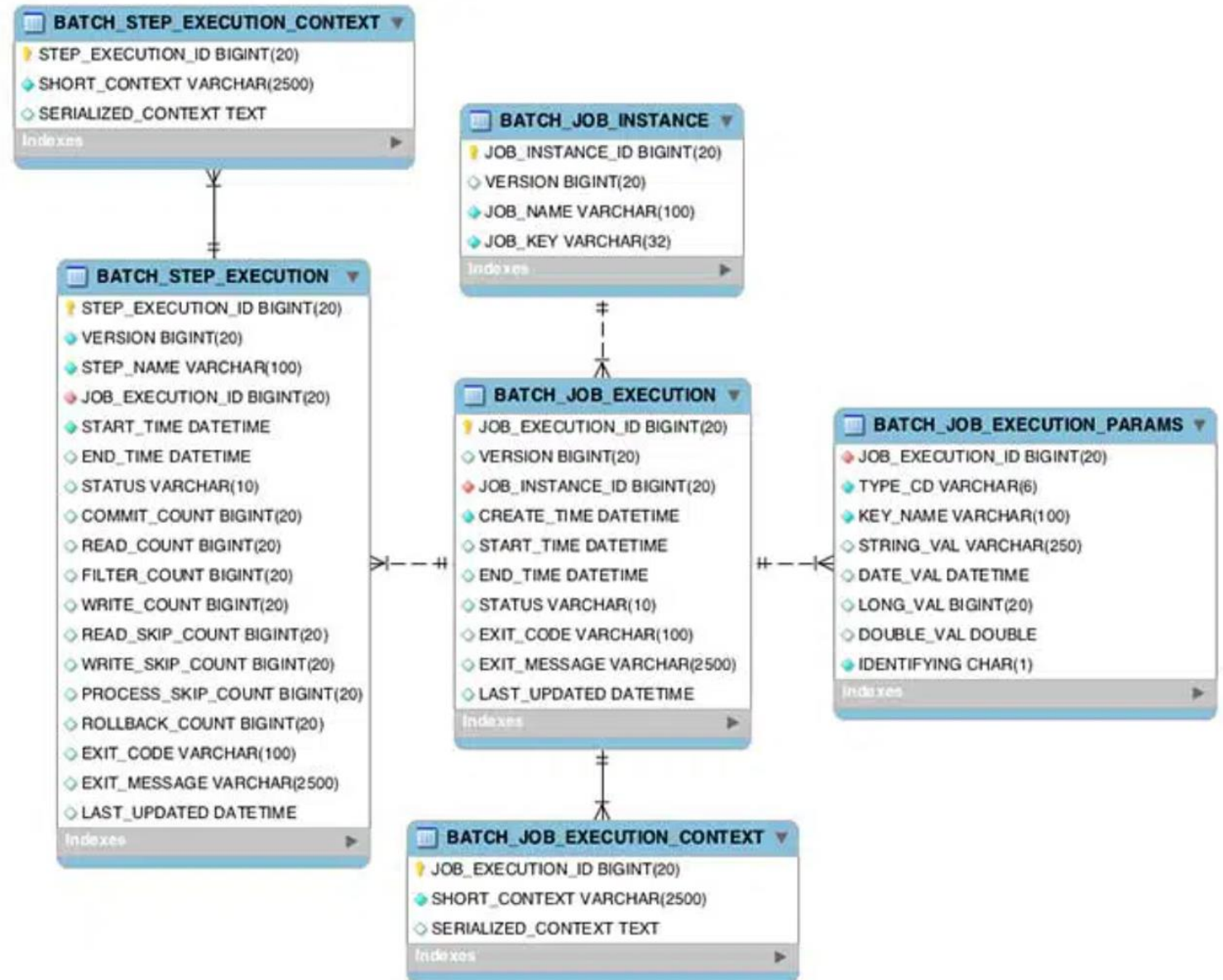
Tema 2

Artefactos esenciales y opcionales .-

- Los artefactos esenciales en Spring Batch son el JobRepository para el almacenamiento de metadatos, el JobLauncher para iniciar trabajos, el Job que representa un proceso completo y los Step que definen los pasos individuales dentro de un Job. Además, dentro de un Step
- Los componentes opcionales pero fundamentales para el procesamiento de datos son el ItemReader (lectura), ItemProcessor (transformación) y ItemWriter (escritura).

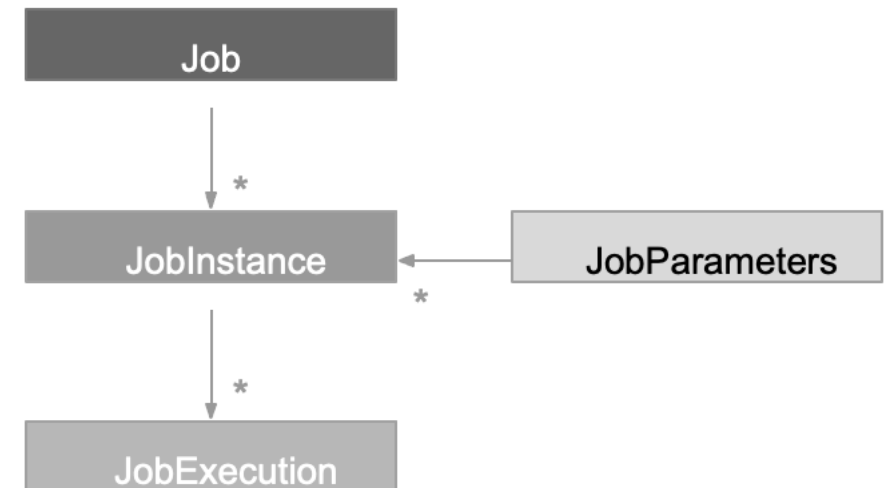
Meta-Data Schema .-

- Una vez se arranca una aplicación Spring Batch, se establece una conexión con la base de datos que contiene el **esquema de tablas** que utiliza el framework.
- Si no existe, se puede incluir por configuración que sea el propio framework el que cree el esquema de base de datos.



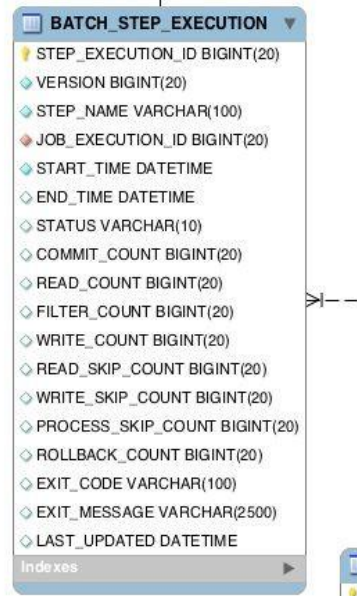
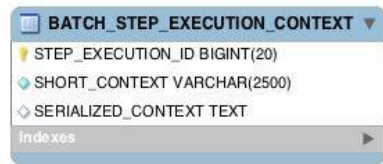
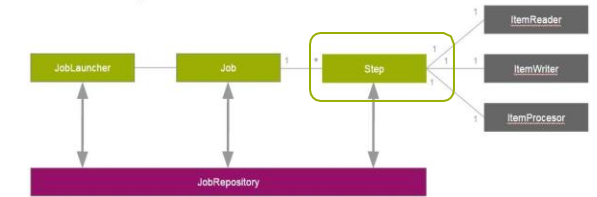
Job (JobInstance, JobParameters, JobExecution) .-

- **Job**
 - El Job es la representación del proceso. Un proceso, a su vez, es un contenedor de pasos (steps).
- **JobInstance**
 - Es una representación lógica de un determinado job con ciertos parámetros de ejecución.
- **JobParameters**
 - Es un conjunto de parámetros utilizado para comenzar la ejecución de un Job. Puede usarse para identificar una ejecución o para proporcionar datos a la propia ejecución.
- **JobExecution**
 - Representa la ejecución de un determinada instancia de un job en un determinado instante de tiempo. Identifica una ejecución del job.



Step (StepExecution) .-

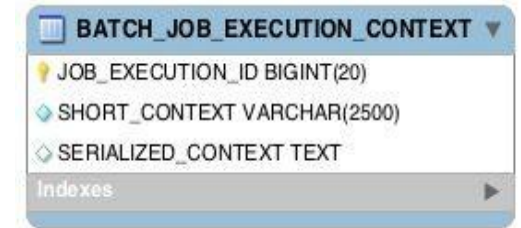
- **Step** encapsula cada una de las fases o pasos de un batch. De este modo un batch está compuesto por uno o más Steps.
- Un Step podrá ser tan simple o complejo o de la tipología que el desarrollador determine oportuno.



- Un **StepExecution** representa cada intento de ejecución de un determinado Step. Cada vez que se ejecuta un Step se creará un nuevo StepExecution.
- Cada StepExecution está formado por un ExecutionContext que contendrá la información que se determine oportuna persistir durante la ejecución del Step como estadísticas o información necesaria del estado del Batch.
- Destacar los siguientes campos relevantes:
 - **Status:** Indica el estado en el que se encuentra un Step. Sus valores variarán entre STARTED, FAILED o COMPLETED.
 - **exitStatus:** Contiene el código de salida del Step.

ExecutionContext .-

- **ExecutionContext** representa una colección de elementos clave/valor controlada por el framework en la que el desarrollador puede persistir información a nivel de Step (StepExecution) o Job (JobExecution).



```
executionContext.putLong(getKey(LINES_READ_COUNT), reader.getPosition());
```

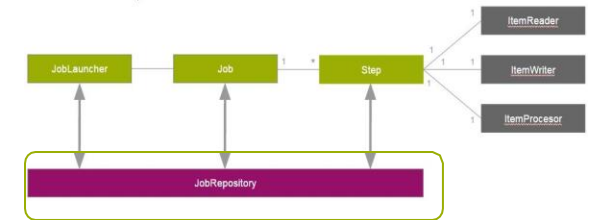
```
long lineCount = executionContext.getLong(getKey(LINES_READ_COUNT));
```

- Se puede recuperar el ExecutionContext en cualquier punto de la ejecución de un batch del siguiente modo:

```
ExecutionContext ecStep = stepExecution.getExecutionContext();  
ExecutionContext ecJob = jobExecution.getExecutionContext();
```

JobRepository .-

- JobRepository es el mecanismo de persistencia para todos los elementos que forman un batch.
- El JobRepository provee de operaciones para la gestión del JobLauncher, Job y los Steps del batch.



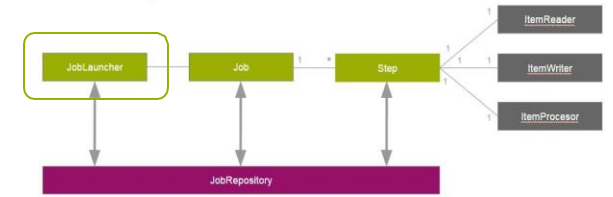
- En el momento en el que un Batch se ejecuta por primera vez, se genera un JobExecution a través del JobRepository y durante su ejecución los datos generados en los StepExecutions y JobExecution se persisten a través del JobRepository.

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

- El JobRepository será el elemento que permitirá persistir la información referente a la ejecución del batch en la base de datos.

JobLauncher .-

- JobLauncher
- JobLauncher representa una simple interfaz para lanzar ejecuciones de un Job con un conjunto de JobParameters como entrada.



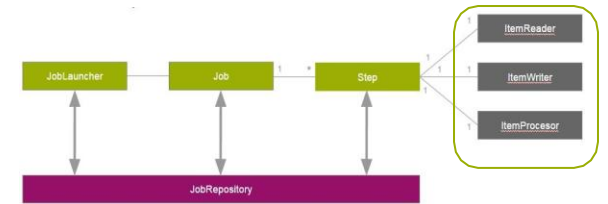
```
public interface JobLauncher {
    public JobExecution run(Job job, JobParameters jobParameters)
        throws JobExecutionAlreadyRunningException, JobRestartException;
}
```

Ejemplo de código

```
try {
    JobExecution execution = jobLauncher.run(job, new
    JobParameters()); System.out.println("Exit Status : " +
    execution.getStatus());
} catch (Exception e) {
    e.printStackTrace();
}
```


Item Readers, Item Writers, Item Processors .-

- **Item Reader**
- Representa la fase de lectura de información para un Step. El ItemReader realizará la lectura elemento a elemento. Una vez concluya la lectura de todos los elementos de la fuente de información configurada retornará null.
- **Item Writer**
- Representa la fase de salida o escritura de información de un Step, batch o chunk. Generalmente un ItemWriter no tiene conocimiento de la información que recibirá a continuación, únicamente del elemento que se encuentra procesando en cada instante.
- **Item Processor**
- Representa la lógica de negocio implementada para realizar el procesamiento de la información. Mientras que un ItemReader realiza la lectura de elementos y el ItemWriter se encarga de la persistencia de la información, ItemProcesor provee de elementos de transformación de la información entre la fase de lectura y posterior fase de escritura. En el caso de que un ItemProcesor retorne null indicará que para dicho elemento no es necesario que se realice la fase de escritura.



Configuración de una tarea (Job)

Tema 3

Configurar el job .-

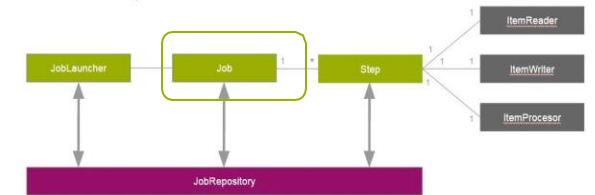
- Un Job aparte de ser un contenedor de Steps, dispone de un gran número de parámetros de configuración.
- La configuración básica de un job será el siguiente:

```
<job id="nombreJob">  
  <step id="nombreStep" parent="refParent" next="nombreStepSiguiente"/>  
  <step id="nombreStepSiguiente" parent="refParent"/>  
</job>
```

- De forma adicional se podrá especificar el otros parámetros:

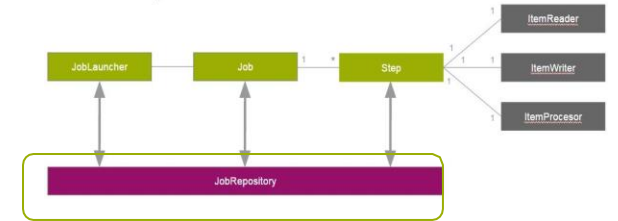
```
<job id="nombreJob" job-repository="specialRepository" restartable="false" parent="jobPadre">  
  <listeners> <listener ref="sampleListener"/></listeners>  
  <validator ref="paremetersValidator"/>.....
```

- JobRepository: Permite especificar el jobRepository al que hace referencia cada Job.
- Restartable: Permite especificar si un job puede reiniciar su ejecución o no.
- Listeners/Interceptors: Permite registrar escuchadores de eventos propios del job (inicio, fin...).
- Parent: Permite especificar un job padre del que hereda sus características de configuración.
- Validator: Permite validar que los parámetros de entrada de un job cumplen ciertas especificaciones.



Configurar el JobRepository .-

- Como se comentaba anteriormente, el JobRepository permitirá el acceso a la base de datos para almacenar la información relativa a la ejecución del batch y dotará de métodos a la infraestructura para gestionar el JobLauncher, el Job y los Steps.



```
<job-repository id="jobRepository" /* Obligatorio - Identificador del objeto que representa el jobRepository */  
  
data-source="dataSource" /* Datasource con acceso a la base de datos */  
transaction-manager="transactionManager"  
isolation-level-for-create="SERIALIZABLE" /* Permite almacenar metadatos del batch para su relanzamiento */ table-  
prefix="BATCH_" /* Define el schema de la base de datos */  
max-varchar-length="1000"/>
```

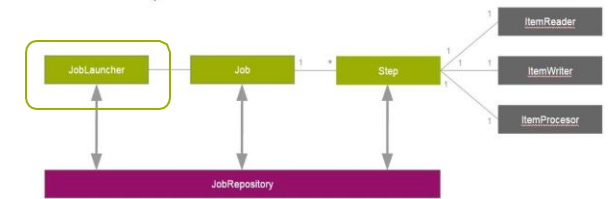
- A través de la definición del jobRepository podremos especificar que su información no sea persistida en base de datos y se almacene en memoria.

```
<bean id="jobRepository" class="org.springframework....MapJobRepositoryFactoryBean">  
  <property name="transactionManager" ref="transactionManager"/>  
</bean>
```

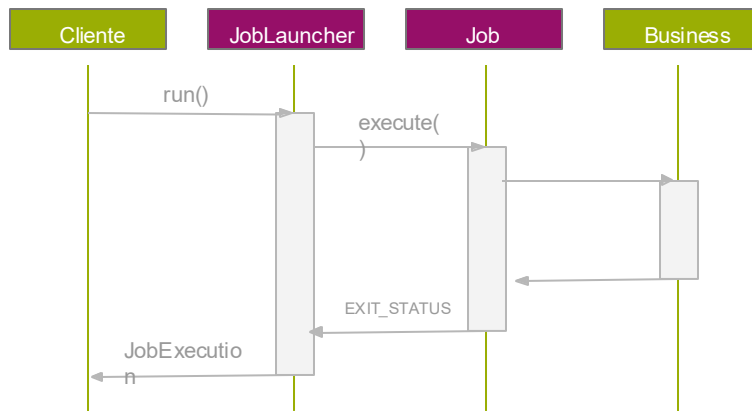
Configurar el JobLauncher .-

- La implementación más básica es la del SimpleJobLauncher ya que únicamente requiere de la referencia al JobRepository para iniciar una ejecución:

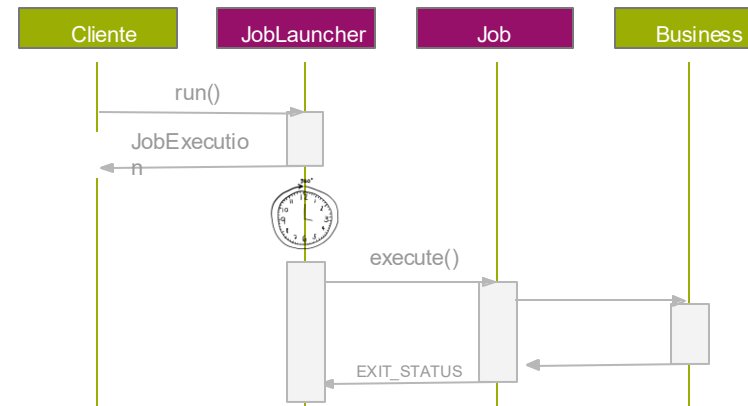
```
<bean id="jobLauncher" class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
  <property name="jobRepository" ref="jobRepository" />
  <property name="taskExecutor"> <!-- Opcional Asíncrono -->
    <bean class="org.springframework.core.task.SimpleAsyncTaskExecutor" />
  </property>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



- Ejecución Síncrona:
- Se espera por el retorno de la ejecución.



- Ejecución asíncrona:
- La ejecución del batch es asíncrona.



Ejecución de un Job .-

- Para poder realizar la ejecución de un Batch se necesitan al menos dos cosas, el JobLauncher y el propio batch a ejecutar. Existen varios modos de realizar la ejecución de un batch, entre ellos los más empleados se encuentra la ejecución desde la línea de comandos y la ejecución desde el propio contexto de ejecución del batch a ejecutar.
- **Ejecución desde la línea de comandos**
 - Opción empleada para aquellos casos en los que se quiera automatizar la ejecución programada de la ejecución de un batch.

```
java -cp "target/dependency-jars/*:target/your-project.jar" org.springframework.batch.core.launch.support.CommandLineJobRunner spring/batch/jobs/job-read-files.xml readJob param1=value1
```

- **Ejecución desde el contexto de ejecución del batch**

```
@Controller
public class JobLauncherController {

    @Autowired
    JobLauncher jobLauncher;

    @Autowired
    Job job

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

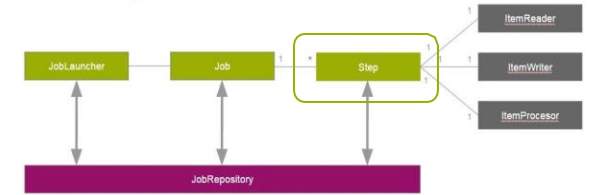
- Opción que permite iniciar la ejecución de un proceso batch mediante una petición HttpRequest, para ello se requiere la creación de un Controlador MVC del modo expuesto.
- El batch se ejecutará de manera asíncrona sin necesidad de que la petición HTTP espere por el retorno de la ejecución del proceso batch.

Configuración de un paso (step) de una tarea

Tema 4

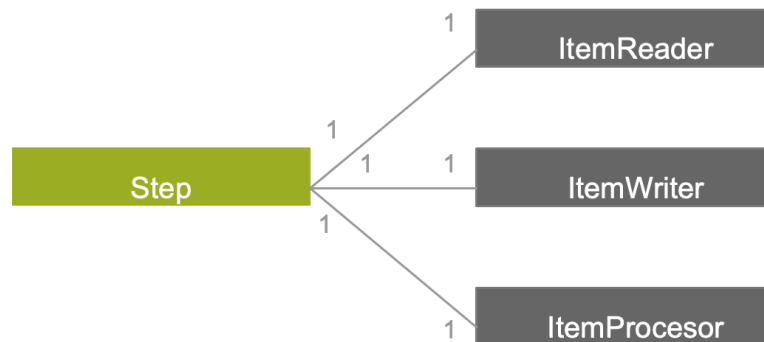
Configurar un Step .-

- Un Step encapsula una fase independiente de funcionalidad y contiene toda la información necesaria para definir y controlar la ejecución del batch. Todo batch debe tener al menos un step.



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

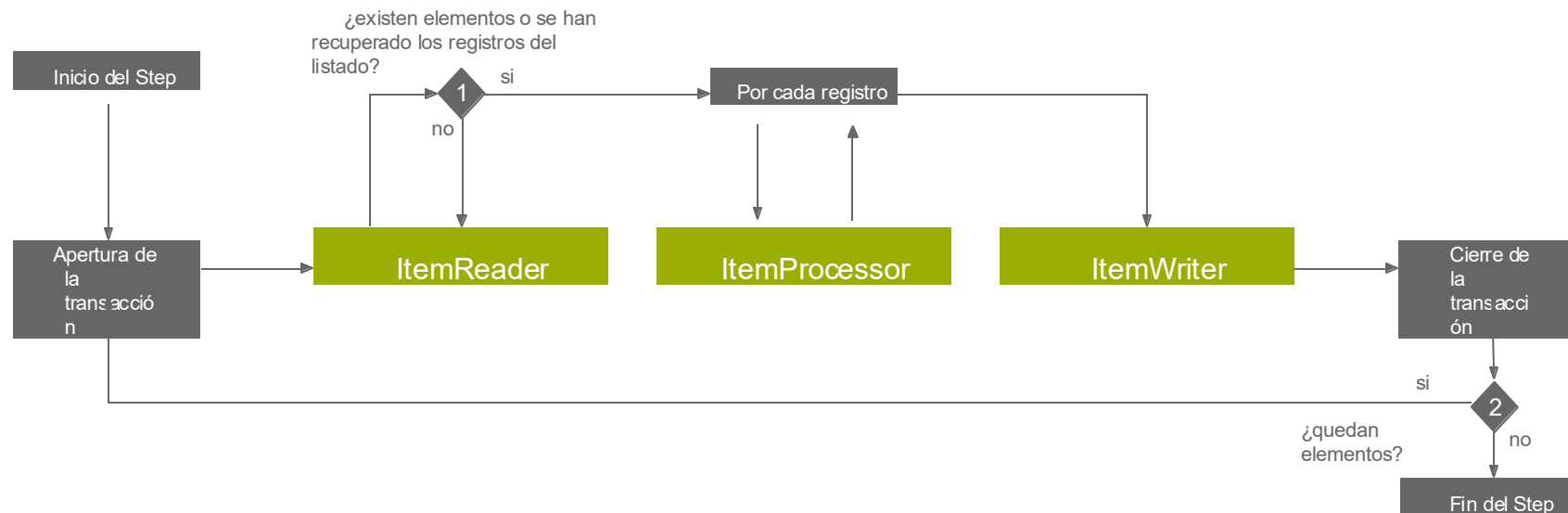
- Un step puede estar compuesto de tres elementos: reader, writer y processor:
- ItemReader: Elemento responsable de leer datos de una fuente de datos (BBDD, fichero, cola de mensajes, etc...)



- ItemProcessor: Elemento responsable de tratar la información obtenida por el reader. No es obligatorio su uso.
- ItemWriter: Elemento responsable de guardar la información leída por el reader o tratada por el processor. Si hay un reader debe haber un writer.

Chunks .-

- Un Chunk se corresponde con la tipología de steps más utilizada en los procesos batch. Consisten en la construcción de un componente especializado en la lectura de elementos (ItemReader), un componente encargado de su procesamiento opcional (ItemProcessor) y un componente que se encarga de la persistencia (ItemWriter).



- Los distintos elementos que constituyen un chunk podrán sobreescribirse para customizarse en función de las necesidades de negocio. El flujo de un chunk se complementará al introducir políticas de reintento y omisión de registros.

Tasklets .-

- Un tasklet es un objeto que contiene cualquier lógica que será ejecutada como parte de un trabajo. Se construyen mediante la implementación de la interfaz Tasklet y son la forma más simple para ejecutar código.
- La interfaz Tasklet contiene únicamente un método execute que será ejecutado repetidamente mientras el retorno del Tasklet sea distinto a RepeatStatus.FINISHED o bien se lance una excepción.

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

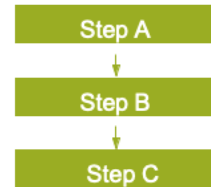
- Se puede emplear el TaskletAdapter para customizar el método al que invocar del siguiente modo:

```
<bean id="myTasklet" class="org.springframework.batch.core.step.tasklet.MethodInvokingTaskletAdapter">  
  <property name="targetObject">  
    <bean class="my.class.CustomTaskletAdapterClass"/>  
  </property>  
  <property name="targetMethod" value="myCustomMethod" />  
</bean>
```

Flujo de Steps .-

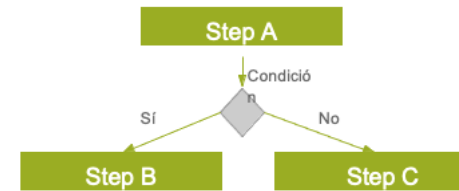
- A través del control de flujo de ejecución de Steps es posible definir lógicas de negocio en función del estado de salida de otros Steps.

Secuencial



```
<job id="job">
  <step id="stepA" next="stepB"/>
  <step id="stepB" next="stepC"/>
  <step id="stepC"/>
</step>
```

Condicional



```
<job id="job">
  <step id="stepA">
    <next on="*" to="stepB"> // * -> 0 o más
    caracteres
    <next on="FAILED" to="stepC">
  </step>
  <step id="stepB"/>
  <step id="stepC"/>
</step>
```

- Otros elementos a tener en cuenta en la definición de flujos entre steps:
 - BatchStatus: Representa el estado de un Job o Step (COMPLETED, STARTED, FAILED,...)
 - ExitStatus: Representa el estado de un Step al finalizar su ejecución.
 - Tag "end": Determina la finalización inmediata del Job tras cumplir su condición. ExitStatus y BatchStatus en estado COMPLETED.
 - Tag "fail": Determina la finalización inmediata del Job tras cumplir su condición. ExitStatus y BatchStatus en estado FAILED.
 - Tag "stop": Determina la parada inmediata del Job tras cumplir su condición. BatchStatus en estado STOPPED.

Scopes (Job/Step) .-

- El Scope permite definir el ámbito en el que se desea crear un bean y en qué momento de la fase de generación de objetos del contenedor de Spring se va a crear permitiendo de este modo definir el orden de generación.
- Es necesario incorporar al XML el siguiente namespace:

```
<beans .... xmlns:batch="http://www.springframework.org/schema/batch"
```

- Step Scope
- Se requiere esta configuración en el caso de que sea necesario que el Step se inicie antes de que se cree la instancia del Bean (carga de propiedades, enlace con base de datos,...)

```
<bean id="step1" scope="step">  
.....  
</bean>
```

- Job Scope
- Sólo permitirá la creación de un determinado bean por job.
- Permitirá recuperar propiedades del job, jobExecutionContext o jobParameters.

```
<bean id="step1" scope="job">  
.....  
</bean>
```

Mecanismos de lectura y escritura

Tema 5

FlatFileItemReaders .-

- Componente genérico de Spring Batch que permite realizar la obtención de información en un fichero o stream.
- Este componente genérico permite configurar los siguientes aspectos de su implementación:

```
<bean id="cvsFileItemReader"
class="org.springframework....file.FlatFileItemReader">
  <property name="resource" value="file:csv/inputs/report.csv" />
  <property name="lineMapper">
    <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
      <property name="lineTokenizer">
        <bean
class="org.springframework.batch....transform.DelimitedLineTokenizer">
          <property name="names" value="id,name" />
        </bean>
      </property>
      <property name="fieldSetMapper">
        <bean class="com.everis.uco.spring.batch....MyObjectFieldSetMapper" />
      </property>
    </bean>
  </property>
</bean>
```

- **Resource:** Recurso de entrada (fichero).
- **LineMapper:** Permite realizar la lectura de información. Se podrán realizar numerosas configuraciones sobre este elemento para determinar el número de campos a obtener, líneas que ignorar,...
- **FieldSetMapper:** Componente que permite realizar el mapeo de la información obtenida a objetos generados con una determinada clase.

FlatFileItemWriters .-

- Componente genérico de Spring Batch que permite realizar la persistencia de información en un fichero o stream.
- Este componente genérico permite configurar los siguientes aspectos de su implementación:

```
<bean id="itemWriter"
class="org.springframework.batch.item.file.FlatFileItemWriter">
  <property name="resource" ref="outputResource" />
  <property name="lineAggregator">
    <bean class="org.spr...FormatterLineAggregator">
      <property name="fieldExtractor">
        <bean
class="org.spr...BeanWrapperFieldExtractor">
          <property name="names" value="name,credit" />
        </bean>
      </property>
      <property name="format" value="%-9s%-2.0f" />
    </bean>
  </property>
</bean>
```

- **Resource:** Recurso de salida (fichero).
- **LineAggregator:** Permite agregar varios campos en una única fila (String). Es el opuesto al LineTokenizer. Implementará el método `aggregate(T item)`.
- **FieldExtractor:** Componente genérico que permite extraer parámetros de un bean. Su utilización junto a `BeanWrapperFieldExtractor` permitirá especificar a través de la propiedad `names` el nombre de los atributos del bean que extraer para poder generar la salida.

XML Item Readers y Writers .-

- Spring Batch facilita utilidades para realizar la lectura y escritura de información en XMLs. A continuación se detalla cómo realizarlo a través del StAX API.

StaxEventItemReader

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

- **fragmentRootElementName:** Elemento padre del XML (root-element).
- **resource:** Acceso al fichero de entrada que contiene la información en formato XML.
- **unmarshaller:** Facilidad OXM que permite realizar el mapeo de los campos definidos en el XML en los campos de objetos Java para su posterior tratamiento y manejo.

```
<bean id="tradeMarshaller"
  class="org.springframework.xml.xstream.XStreamMarshaller">
  <property name="aliases">
    <util:map id="aliases">
      <entry key="trade" value="org.springframework.batch.sample.domain.Trade" />
      <entry key="price" value="java.math.BigDecimal" />
      <entry key="name" value="java.lang.String" />
    </util:map>
  </property>
</bean>
```

StaxEventItemWriter

```
<bean id="itemWriter" class="org.springframework.batch.item.xml.StaxEventItemWriter">
  <property name="resource" ref="outputResource" />
  <property name="marshaller" ref="customerCreditMarshaller" />
  <property name="rootTagName" value="customers" />
  <property name="overwriteOutput" value="true" />
</bean>
```

- **rootTagName:** Elemento padre del XML (root-element).
- **resource:** Acceso al fichero de entrada que contiene la información en formato XML.
- **marshaller:** Facilidad OXM que permite realizar el mapeo de los campos de los objetos Java en los campos del XML.
- **overwriteOutput:** Sobreescribe el fichero de salida en caso de existir.

```
<bean id="customerCreditMarshaller"
  class="org.springframework.xml.xstream.XStreamMarshaller">
  <property name="aliases">
    <util:map id="aliases">
      <entry key="custom"
        value="org.springframework.batch.sample.domain.CustomerCredit" />
      <entry key="credit" value="java.math.BigDecimal" />
      <entry key="name" value="java.lang.String" />
    </util:map>
  </property>
</bean>
```


Entrada desde varios ficheros .-

- Es un requisito habitual procesar varios ficheros como entrada de un único Step. Si asumimos que todos los ficheros tienen el mismo formato, MultiResourceItemReader permite realizar este tipo de entrada tanto para XML como para un FlatFileItemReader.

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

- Este ejemplo se apoyará en el uso de un FlatFileItemReader. Esta configuración de entrada para ambos ficheros, maneja tanto el rollback como el reinicio del step de manera controlada.

- Se recomienda que cada Job trabaje con su propio directorio de forma individual hasta que se complete la ejecución.

Database ItemReaders y ItemWriters .-

- En la mayoría de sistemas corporativos, los datos se alojan en sistemas de persistencia basados en bases de datos. A continuación se detallan los principales componentes disponibles:

```
<bean id="itemReader" class="org.spr...JdbcCursorItemReader">
  <property name="dataSource" ref="dataSource" />
  <property name="sql" value="select ID, NAME, CREDIT from CUSTOMER" />
  <property name="rowMapper"> <bean class="org.springframework.batch.sample.domain.CustomerCreditRowMapper" /> </property>
</bean>
```

ItemReader	Descripción
JdbcCursorItemReader	Lee de un cursor de base de datos a través de JDBC.
HibernateCursorItemReader	Lee de un cursor de base de datos a través de HQL.
StoredProcedureItemReader	Lee de un cursor de base de datos a través de un proceso almacenado (ej: PL/SQL).
JdbcPagingItemReader	A partir de una sentencia SQL, pagina los resultados que pueden leerse sin verse afectada la memoria del proceso ante grandes volúmenes de datos.
JpaPagingItemReader	A partir de una sentencia JSQL, pagina los resultados que pueden leerse sin verse afectada la memoria del proceso ante grandes volúmenes de datos.
IbatisPagingItemReader	A partir de una sentencia iBATIS, pagina los resultados que pueden leerse sin verse afectada la memoria del proceso ante grandes volúmenes de datos.
HibernatePagingItemReader	Lee a partir de una sentencia HQL paginada.
MongoItemReader	A partir de un operador de mongo y una sentencia JSON válida de MongoDB, realiza la lectura de elementos de la base de datos.

Database ItemReaders y ItemWriters .-

- Los ItemWriters definirán el modo en el que la información tras ser procesada será almacenada en los sistemas de persistencia.

```
<bean id="databaseItemWriter" class="org.springframework.batch.item.database.JdbcBatchItemWriter">
  <property name="dataSource" ref="dataSource" />
  <property name="sql">
    <value>
      <![CDATA[ insert into EXAM_RESULT(STUDENT_NAME, DOB, PERCENTAGE) values (?, ?, ?)]]>
    </value>
  </property>
  <property name="itemPreparedStatementSetter">
    <bean class="com.everis.....CustomItemSetter" />
  </property>
</bean>
```

ItemWriter	Descripción
HibernateItemWriter	Utiliza una sesión de hibernate para manejar la transaccionalidad de la persistencia de la información.
JdbcBatchItemWriter	Utiliza sentencias de tipología PreparedStatement y puede utilizar steps rudimentarios para localizar fallos en la persistencia de la información.
JpaItemWriter	Utiliza un EntityManager de JPA para poder manejar la transaccionalidad en la persistencia de la información.
MongoItemWriter	A partir de un objeto de tipo MongoOperations, permite realizar la persistencia de la información en bases de datos MongoDB. La escritura de la información se retrasa hasta el último momento antes de realizar la validación de la persistencia de la información.

ItemReaders y ItemWriters customizados .-

- Una vez vistos los distintos componentes genéricos facilitados por Spring Batch, hay muchos escenarios que pueden no estar cubiertos por estas implementaciones. En estos casos podremos crear nuestras implementaciones customizadas de ItemReader e ItemWriter.

Custom ItemReader

- Bean que realizará la obtención de la información cuya implementación será definida por el usuario.
- Implementa la interfaz ItemReader<T> cuyo método read que obtendrá los objetos a tratar durante la fase de escritura.

```
public class CustomReader implements
ItemReader<MyObject> {
    @Override
    public MyObject read() throws Exception, UnexpectedInputException,
ParseException, NonTransientResourceException {
        TODO Auto-generated method stub
        return null;
    }
}
```

Custom ItemWriter

- Bean que realizará la persistencia de la información cuya implementación será definida por el usuario.
- Implementa la interfaz ItemWriter<T> cuyo método write recibirá el listado de objetos a tratar durante la fase de escritura.

```
public class CustomWriter implements
ItemWriter<MyObject> {
    @Override
    public void write(List<? extends MyObject> items) throws Exception {
        TODO Auto-generated method stub
    }
}
```

Escalabilidad y concurrencia

Tema 6

Steps multihilo .-

- Para configurar la ejecución de un mismo step por varios hilos la forma más simple es la creación de un pool de hilos mediante la configuración de un TaskExecutor. Su definición se realizará del siguiente modo:

```
<step id="step1">  
  <tasklet  
    task-  
    executor="taskExecutor"  
    throttle-limit="20">  
    ....  
    ....  
  </tasklet>  
</step>
```

- La implementación del objeto "taskExecutor" podrá cualquier implementación de la interfaz TaskExecutor, por ejemplo, SimpleAsyncTaskExecutor.
- En este caso, cada hilo realizará la ejecución del mismo step de forma independiente, pudiendo realizarse el procesamiento de elementos de manera no consecutiva. En algunas situaciones será necesario limitar el número de hilos, para ello se especificará el parámetro throttle-limit.

- **IMPORTANTE:** Verificar que los componentes utilizados sean "thread safe" y se puedan utilizar en steps multihilo.

Steps paralelos .-

- En la definición de la estructura de determinados batchs es posible identificar cierta lógica u operativa que es necesaria paralelizar. Para ello es posible particionar y delegar responsabilidades de la operativa asignándoles steps individuales que poder paralelizar en un único proceso. La configuración necesaria para poder paralelizar steps sería la siguiente:

```
<job id="job1">
  <split id="split1" task-executor="taskExecutor" next="step4">
    <flow>
      <step id="step1" parent="s1" next="step2"/>
      <step id="step2" parent="s2"/>
    </flow>
    <flow>
      <step id="step3" parent="s3"/>
    </flow>
  </split>
</job>
<beans:bean id="taskExecutor" class="org.spr...SimpleAsyncTaskExecutor"/>
```

- Como se puede ver en el código, es necesario realizar la definición de un elemento "taskExecutor" que hace referencia a la implementación del TaskExecutor a emplear para ejecutar cada uno de los flujos de trabajo.
- SyncTaskExecutor es la implementación por defecto de TaskExecutor.

- El job no finalizará su estado como completo hasta que puede agregar el estado de salida de cada uno de los flujos.

Gestión de errores

Tema 7

Estrategias principales de manejo de errores .-

- Retry (Reintentos)
- Skip (Omitir elementos)
- Restart (Reinicio de trabajos)
- Recovery (Recuperacion)

Retray .-

- El mecanismo de retry permite reintentar operaciones fallidas un número determinado de veces antes de considerar que el elemento ha fallado definitivamente

```
@Bean
public Step retryStep() {
    return stepBuilderFactory.get("retryStep")
        .<Input, Output>chunk(10)
        .reader(itemReader())
        .processor(itemProcessor())
        .writer(itemWriter())
        .faultTolerant()
        .retryLimit(3) // Máximo 3 intentos
        .retry(DeadlockLoserDataAccessException.class)
        .retry(TransientDataAccessException.class)
        .build();
}
```

Skip .-

- El mecanismo de skip permite omitir elementos problemáticos y continuar procesando el resto del lote.

```
@Bean
public Step skipStep() {
    return stepBuilderFactory.get("skipStep")
        .<Input, Output>chunk(10)
        .reader(itemReader())
        .processor(itemProcessor())
        .writer(itemWriter())
        .faultTolerant()
        .skipLimit(10) // Máximo 10 elementos omitidos
        .skip(FlatFileParseException.class)
        .skip(ValidationException.class)
        .noSkip(NullPointerException.class) // Nunca omitir este tipo
        .build();
}
```

Restart .-

- Spring Batch permite reiniciar trabajos fallidos desde el punto exacto donde fallaron, sin reprocesar datos ya completados.

```
@Bean
public Job restartableJob() {
    return jobBuilderFactory.get("restartableJob")
        .incrementer(new RunIdIncrementer())
        .start(step1())
        .next(step2())
        .build();
}
```

```
@Bean
public Step restartableStep() {
    return stepBuilderFactory.get("restartableStep")
        .<Input, Output>chunk(100)
        .reader(itemReader())
        .processor(itemProcessor())
        .writer(itemWriter())
        .allowStartIfComplete(true) // Permitir reinicio
        .build();
}
```

Recovery .-

- Cuando se agotan todos los reintentos, se puede definir una lógica de recuperación para manejar elementos fallidos.

```
@Component
public class RecoveryService {

    @Retryable(value = {SQLException.class}, maxAttempts = 3)
    public void processData(DataItem item) throws SQLException {
        // Lógica de procesamiento que puede fallar
        dataRepository.save(item);
    }

    @Recover
    public void recover(SQLException ex, DataItem item) {
        // Lógica de recuperación cuando se agotan los reintentos
        errorRepository.saveFailedItem(item, ex.getMessage());
        notificationService.sendAlert("Failed to process item: " + item.getId());
    }
}
```

Listeners para gestión de errores .-

- Spring Batch
proporciona interfaces de listener para capturar y manejar errores en diferentes puntos del procesamiento

```
@Component
public class CustomStepListener implements StepExecutionListener {

    private static final Logger logger = LoggerFactory.getLogger(CustomStepListener.class);

    @Override
    public void beforeStep(StepExecution stepExecution) {
        logger.info("Iniciando step: {}", stepExecution.getStepName());
    }

    @Override
    public ExitStatus afterStep(StepExecution stepExecution) {
        List<Throwable> failures = stepExecution.getFailureExceptions();

        if (!failures.isEmpty()) {
            logger.error("Step {} falló con {} excepciones",
                stepExecution.getStepName(), failures.size());

            // Registrar cada excepción
            failures.forEach(ex -> logger.error("Error: ", ex));

            // Decidir si continuar o fallar
            if (failures.size() > 5) {
                return ExitStatus.FAILED;
            }
        }

        logger.info("Step completado. Read: {}, Write: {}, Skip: {}",
            stepExecution.getReadCount(),
            stepExecution.getWriteCount(),
            stepExecution.getSkipCount());

        return ExitStatus.COMPLETED;
    }
}
```

Testing

Tema 8

Testing en Spring Batch .-

- Para garantizar que tus trabajos batch funcionen correctamente y manejen errores según lo previsto, es fundamental implementar pruebas tanto a nivel de step como de job, aprovechando el soporte de Spring Batch Test y Spring Boot Test

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

<dependency>
  <groupId>org.springframework.batch</groupId>
  <artifactId>spring-batch-test</artifactId>
  <scope>test</scope>
</dependency>
```


Testing en Spring Batch.-

```
@SpringBootTest
@AutoConfigureTestDatabase
@TestExecutionListeners({ DependencyInjectionTestExecutionListener.class,
                          StepScopeTestExecutionListener.class })
class SkipRetryStepTest {

    @Autowired
    private StepBuilderFactory steps;

    @Autowired
    private JobRepository jobRepository;

    @Autowired
    private PlatformTransactionManager transactionManager;

    // Bean del step que quieres probar
    @Autowired
    @Qualifier("skipStep")
    private Step skipStep;

    @Test
    void testSkipPolicy() throws Exception {
        JobExecution jobExec = jobRepository.createJobExecution("testJob", new JobParametersBuilder().addJobParameters());
        StepExecution stepExec = jobExec.createStepExecution("skipStep");

        // Ejecutar el step aislado
        RepeatStatus status = StepScopeTestUtils.doInStepScope(stepExec, () ->
            skipStep.execute(stepExec)
        );

        assertEquals(BatchStatus.COMPLETED, stepExec.getStatus());
        assertTrue(stepExec.getSkipCount() > 0, "Debe haber elementos omitidos");
    }
}
```

Spring Batch



Completa nuestra encuesta
de satisfacción a través del QR



GRACIAS

indra