Breve ejemplo con Spring Batch

Este post muestra un ejemplo simple de uso de Spring Batch. *Spring Batch* es un componente más de la suite de Spring que en este caso nos permite implementar procesamientos batch de forma rápida y sencilla.

Llamaremos procesamiento batch al mecanismo que nos permite ejecutar un conjunto de operaciones similares pero de contenido diferente por lotes, es decir, como un todo, considerando el conjunto de operaciones como una única operación.

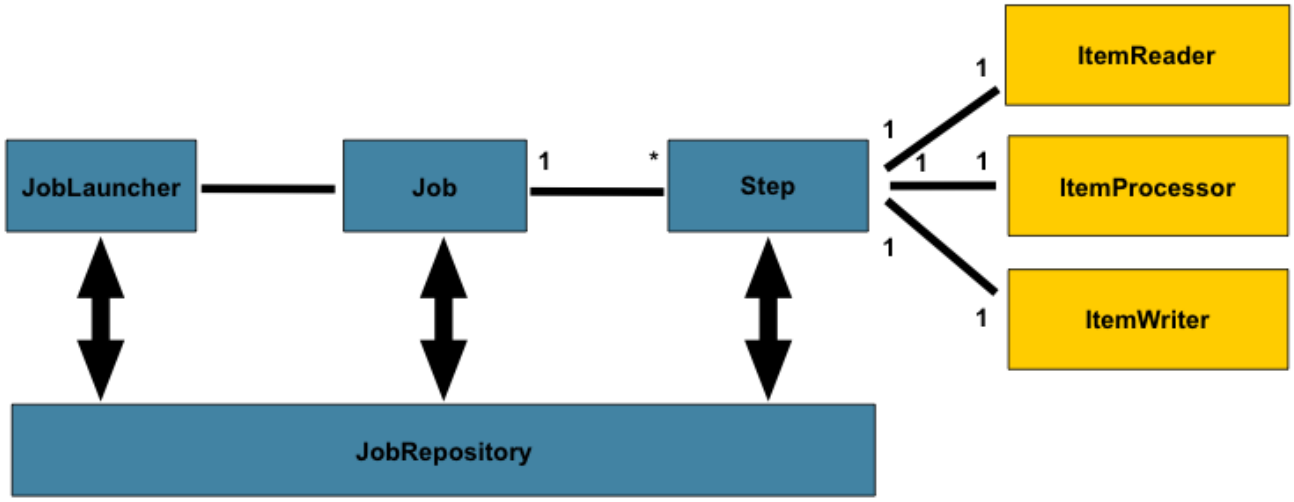
Para ello, *Spring Batch* hace uso de una metodología común a múltiples implementaciones de ejecuciones batch o por lotes. Esta metodología hace uso de los siguientes conceptos para definir y desarrollar los lotes a ejecutar:

- **Job**. Es el mecanismo principal de ejecución Sería la tarea encargada de ejecutar el conjunto de lotes programado. Una ventaja del job es que se puede programar para ejecutarse de forma recurrente, aunque este mecanismo no esta diseñado para sustituir a otros mecanismos o tecnologías de ejecución recurrente y programación como Quartz o similar, mucho más potentes y funcionales que el mecanismo por defecto de estos jobs. Un job estaría formado por steps o pasos de ejecución. Cada uno de esos pasos sería un lote. El job definiría el orden de ejecución de esos pasos o lotes.

- **Step**. Es la representación de un lote. Formaría parte de un job que se encargaria de ejecutarlo, bien en solitario o como parte de una ejecución múltiple de varios lotes diferentes. El step definiría una entrada, en forma de un Reader que leería los datos a procesar, un Processor, que procesaría los datos, es decir, implementaría la transformación del dato en sí, y un Writer que persistiría el procesamiento.

- **Readers, processors y writers**. Como se indica en el anterior punto, serían parte de cada paso y constituirían el flujo de datos de cada lote. El reader leería los datos uno a uno o por bloques, le pasaría los datos al processor que los transformaría, y estos irían despues al writer que, por bloques, persistiría los cambios. Todo el bloque procesado sería considerado una sola transacción.

- Otros elementos a tener en cuenta serían el lanzador de jobs, el respositorio donde se almacena la configuración y el estado de cada job, etc. pero en este ejemplo nos centraremos en los tres primero elementos antes mencionados.



Spring Batch

En este ejemplo vamos a desarrollar una aplicación enterprise que al arrancar ejecuta automaticamente un job con un solo paso que lee de base de datos tres registros, los modifica, y guarda los cambios. Despues, desde una página web, podemos verificar que se hicieron los cambios, y también desde los ficheros de log.

1. Spring Boot y dependencias Maven

Partimos del ejemplo con Spring Data de este otro post. Añadimos la siguiente dependencia al pom.xml:

```
<!-- Add typical dependencies for a spring batch -->
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-batch</artifactId>
</dependency>
```

Esta dependencia incluye toda la parafernalia necesaria para usar *Spring Batch*.

2. Configuración del batch

Configuraremos el batch usando una clase java. Creamos la siguiente clase:

```
/**
* @author lagarcia
*
*/
@Configuration
@EnableBatchProcessing
public class BatchConfiguration {

/** Entity manager factory */
@Autowired
private EntityManagerFactory emFactory;

/**
* Superhero item reader.
*
* @return The superhero item
*/
@Bean
public ItemReader<Superhero> reader() {
```

```
20    JpaPagingItemReader<Superhero> ireader = new JpaPagingItemReader<>();
```

Esto es todo lo que se necesita para configurar el batch. Vamos paso a paso:

- Usamos **@Configuration** para indicar que es una clase de configuración. Después añadimos **@EnableBatchProcessing** para indicar que se configura *Spring Batch*.

- Enlazamos con el *EntityManagerFactory* que usaremos para leer y escribir de la base de datos embebida. *Spring Boot* ha configurado ya por defecto el datasource y lo ha enlazado con el EntityManagerFactory.

```
1  /** Entity manager factory */
2  @Autowired
3  private EntityManagerFactory emFactory;
```

- Definimos el job. En este caso el job tiene un solo paso (*s1*). Usamos el *RunIdIncrementer()* para asignarle un ID incremental a la ejecución del job, ya que *Spring Batch* almacena el estado de cada job en base de datos. También se le asocia un listener que estará escuchando a los eventos del job. Hablaremos de él más adelante.

```
1  /**
2  * Spring batch job.
3  *
4  * @param jobs
5  * @param s1
6  * @param listener
7  * @return
8  */
9  @Bean
10 public Job superHeroJob(JobBuilderFactory jobs, Step s1,
11 JobExecutionListener listener) {
12   return jobs.get("superHeroJob").incrementer(new RunIdIncrementer())
13     .listener(listener).flow(s1).end().build();
14 }
```

- Definimos el lote, en este caso el paso que se ejecutará dentro del job. Especificamos que se escribirán los datos de diez en diez con el método chunk(). El lote o paso tendra un reader, un processor y un writer, como es habitual.

```
1  /**
2  * Spring batch superhero job - Step 01
3  *
4  * @param stepBuilderFactory
5  * @param reader
6  * @param writer
7  * @param processor
8  * @return
9  */
10 @Bean
11 public Step step1(StepBuilderFactory stepBuilderFactory,
12 ItemReader<Superhero> reader, ItemWriter<Superhero> writer,
13 ItemProcessor<Superhero, Superhero> processor) {
14   return stepBuilderFactory.get("step01")
15     .<Superhero, Superhero> chunk(10).reader(reader)
16     .processor(processor).writer(writer).build();
17 }
```

- Definimos el reader. En este caso hacemos uso de un reader para JPA, ya que leemos de una base de datos y tenemos el modelo de datos definido con JPA. Usamos una query en JPQL y especificamos que se pagine lo leido de cinco en cinco elementos (*setPageSize*), que iran pasando al procesador.

```
/**
* Superhero item reader.
*
* @return The superhero item
*/
@Bean
public ItemReader<Superhero> reader() {
  JpaPagingItemReader<Superhero> ireader = new JpaPagingItemReader<>();
  ireader.setEntityManagerFactory(this.getEmFactory());
  ireader.setQueryString("select entity from Superhero entity order by entity.name desc");
  ireader.setPageSize(5);
  return ireader;
}
```

- Definimos el processor. Creamos una nueva clase (SuperheroProcessor.java) para ello. El processor recibe un objeto Superhero y devuelve también un Superhero (podría devolver otro tipo de dato). En este caso el procesador modifica un campo del POJO en función de su ID (name).

```
/**
* Superhero item processor.
*
* @return The superhero processor
*/
@Bean
public ItemProcessor<Superhero, Superhero> processor() {
  return new SuperheroProcessor();
}
```

- Definimos el writer. Simplemente escribira de diez en diez en base de datos y hara commit sobre ese bloque de datos.

```
/**
* Superhero item writer.
*
* @param dataSource
*            - The datasource
* @return The superhero item writer
*/
@Bean
public ItemWriter<Superhero> writer(DataSource dataSource) {
  JpaItemWriter<Superhero> iwriter = new JpaItemWriter<>();
  iwriter.setEntityManagerFactory(this.getEmFactory());
  return iwriter;
}
```

3. Procesamiento de los datos

La clase que implementa el procesador seria tal que así:

```
/**
* @author lagarcia
*
*/
public class SuperheroProcessor implements ItemProcessor<Superhero, Superhero> {

/** A logger reference */
private static Logger logger = Logger.getLogger(SuperheroProcessor.class);

/**
* Process superhero.
*/
@Override
public Superhero process(Superhero superheroArg) throws Exception {
```

```
15    if (superheroArg != null) {
16      logger.info("Procesando a " + superheroArg.getName());
17      switch (superheroArg.getName()) {
18        case "Superman": {
19          superheroArg.setWeakness("Kryptonite");
20          break;
21        }
22        case "Flash": {
23          superheroArg.setWeakness("Speed of light");
24          break;
25        }
26        case "Dr. Magneto": {
27          superheroArg.setWeakness("Plastic");
28          break;
29        }
30      }
31
32      return superheroArg;
33    }
34    return null;
35  }
36  }
```

Como se ha mencionado antes, recibe un objeto de tipo Superhero, lo modifica, y devuelve el mismo objeto.

4. Listener del job

Se define un listener para el job que escucha los eventos del mismo. En este caso se implementa el que afecta a la finalización del job para que muestre un mensaje en el log:

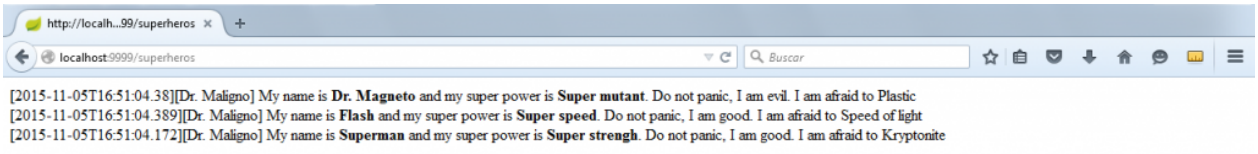
```
1  /**
2  * @author lagarcia
3  *
4  */
5  @Component
6  public class BatchListener extends JobExecutionListenerSupport {
7
8  /** A logger reference */
9  private static Logger logger = Logger.getLogger(BatchListener.class);
10
11 /**
12 * After job execution.
13 */
14 @Override
15 public void afterJob(JobExecution jobExecution) {
16   if (jobExecution.getStatus() == BatchStatus.COMPLETED) {
17     logger.info("¡¡Job " + jobExecution.getJobId() + " finalizado!!");
18   }
19 }
20 }
```

5. Ejecución

Al lanzar la aplicación podemos ver en los logs que el job se ejecutó correctamente (y tb al acceder a los datos desde la página web).

```
INFO : org.springframework.batch.core.launch.support.SimpleJobLauncher - Job: [FlowJob: [name=superHeroJob]] launched with the followi
INFO : org.springframework.batch.core.job.SimpleStepHandler - Executing step: [step01]
INFO : net.luisalbertogh.springbatch.batch.SuperheroProcessor - Procesando a Superman
INFO : net.luisalbertogh.springbatch.batch.SuperheroProcessor - Procesando a Flash
INFO : net.luisalbertogh.springbatch.batch.SuperheroProcessor - Procesando a Dr. Magneto
INFO : net.luisalbertogh.springbatch.batch.BatchListener - ¡¡Job 0 finalizado!!
INFO : org.springframework.batch.core.launch.support.SimpleJobLauncher - Job: [FlowJob: [name=superHeroJob]] completed with the follow
```

Logs de Spring Batch



GUI web

Se puede descargar el proyecto completo desde aquí. Y también en Github.

This entry was posted in Java, Spring and tagged Batch, Spring on November 5, 2015 [http://ictblog.luisalbertogh.net/?p=469] by Chef.