Spring Batch

Contenidos

- Spring con Spring Boot
- Arquitectura.
- Artefactos esenciales.
- Configuración de un trabajo (Job).
- Configuración de un paso (step) de un trabajo.
- Mecanismos de lectura y escritura.
- Escalabilidad y concurrencia.
- Gestión de errores.
- Testing.

http://spring.io

SPRING CON SPRING BOOT

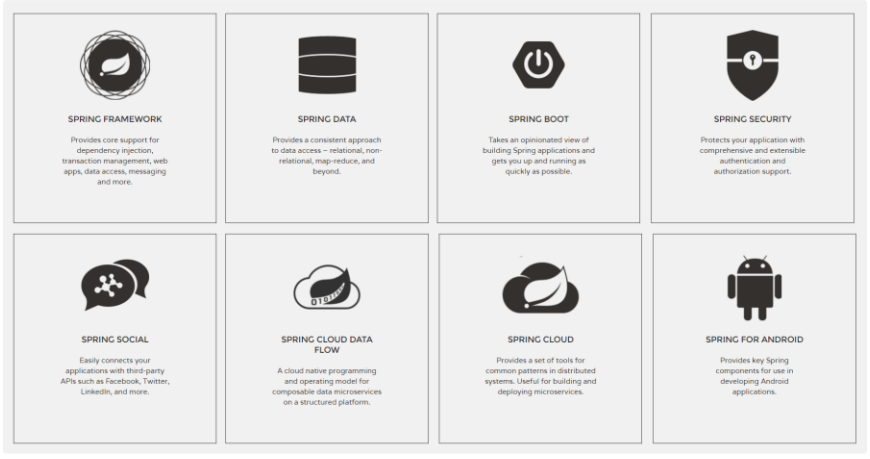
Spring

- Inicialmente era un ejemplo hecho para el libro “J2EE design and development” de Rod Johnson en 2003, que defendía alternativas a la “visión oficial” de aplicación JavaEE basada en EJBs.
- Actualmente es un framework open source que facilita el desarrollo de aplicaciones java JEE & JSE (no esta limitado a aplicaciones Web, ni a java pueden ser .NET, Silverlight, Windows Phone, etc.)
- Provee de un contenedor encargado de manejar el ciclo de vida de los objetos (beans) para que los desarrolladores se enfoquen a la lógica de negocio. Permite integración con diferentes frameworks.
- Surge como una alternativa a EJB’s
- Actualmente es un framework completo compuesto por múltiples módulos/proyectos que cubre todas las capas de la aplicación, con decenas de desarrolladores y miles de descargas al día
  - MVC
  - Negocio (donde empezó originalmente)
  - Acceso a datos

Características

- Ligero
  - No se refiere a la cantidad de clases sino al mínimo impacto que se tiene al integrar Spring.
- No intrusivo
  - Generalmente los objetos que se programan no tienen dependencias de clases específicas de Spring
- Flexible
  - Aunque Spring provee funcionalidad para manejar las diferentes capas de la aplicación (vista, lógica de negocio, acceso a datos) no es necesario usarlo para todo. Brinda la posibilidad de utilizarlo en la capa o capas que queramos.
- Multiplataforma
  - Escrito en Java, corre sobre JVM

Proyectos



Módulos necesarios

- Spring Framework
  - Spring Core
    - Contenedor IoC (inversión de control) - inyector de dependencia
  - Spring MVC
    - Framework basado en MVC para aplicaciones web y servicios REST
- Spring Data
  - Simplifica el acceso a los datos: JPA, bases de datos relacionales / NoSQL, nube
- Spring Boot
  - Simplifica el desarrollo de Spring: inicio rápido con menos codificación

Spring Boot

- Spring tiene una gran cantidad de módulos que implican multitud de configuraciones. Estas configuraciones pueden requerir mucho tiempo, pueden ser desconocidas para principiantes y suelen ser repetitivas.
- La solución de Spring es Spring Boot, que aplica el concepto de Convention over Configuration (CoC).
- CoC es un paradigma de programación que minimiza las decisiones que tienen que tomar los desarrolladores, simplificando tareas.
- No obstante, la flexibilidad no se pierde, ya que a pesar de otorgar valores por defecto, siempre se puede configurar de forma extendida.
- De esta forma se evita la repetición de tareas básicas a la hora de construir un proyecto.
- Spring Boot es una herramienta que nace con la finalidad de simplificar aun más el desarrollo de aplicaciones basadas en el framework Spring Core: que el desarrollador solo se centre en el desarrollo de la solución, olvidándose por completo de la compleja configuración que actualmente tiene Spring Core para poder funcionar.

Spring Boot

- Configuración:
  - Spring Boot cuenta con un complejo módulo que autoconfigura todos los aspectos de nuestra aplicación para poder simplemente ejecutar la aplicación, sin tener que definir absolutamente nada.
- Resolución de dependencias:
  - Con Spring Boot solo hay que determinar que tipo de proyecto estaremos utilizando y el se encarga de resolver todas las librerías/dependencias para que la aplicación funcione.
- Despliegue:
  - Spring Boot se puede ejecutar como una aplicación Stand-alone, pero también es posible ejecutar aplicaciones web, ya que es posible desplegar las aplicaciones mediante un servidor web integrado, como es el caso de Tomcat, Jetty o Undertow.

Spring Boot

- Métricas:
  - Por defecto, Spring Boot cuenta con servicios que permite consultar el estado de salud de la aplicación, permitiendo saber si la aplicación está encendida o apagada, memoria utilizada y disponible, número y detalle de los Bean's creado por la aplicación, controles para el prendido y apagado, etc.
- Extensible:
  - Spring Boot permite la creación de complementos, los cuales ayudan a que la comunidad de Software Libre cree nuevos módulos que faciliten aún más el desarrollo.
- Productividad:
  - Herramientas de productividad para desarrolladores como Spring Initializr, Lombok, LiveReload y Auto Restart, funcionan en su IDE favorito: Spring Tool Suite, IntelliJ IDEA y NetBeans.

Con Eclipse

- Descargar Hibernate:
  - http://hibernate.org/orm/downloads/
- Descargar e instalar JDK:
  - http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
- Descargar y descomprimir Eclipse:
  - https://www.eclipse.org/downloads/
- Añadir a Eclipse las Hibernate Tools
  - Help > Eclipse Marketplace: JBoss Tools
- Crear una User Librarie para Hibernate
  - Window > Preferences > Java > Build Path > User Libraries > New
  - Add External JARs: \lib\required
- Descargar y registrar la definición del driver JDBC
  - Window > Preferences > Data Management > Connectivity > Driver Definition > Add

Instalación Spring Tool Suite

- https://spring.io/tools
- Spring Tool Suite
  - IDE gratuito, personalización del Eclipse
- Plug-in para Eclipse (VSCode, Atom)
  - Help → Eclipse Marketplace …
    - Spring Tools 4 for Spring Boot

devtools

- Al realizar nuevos cambios en nuestra aplicación, podemos hacer que el arranque se reinicie automáticamente. Para eso es necesario incluir una dependencia Maven extra: spring-boot-devtools.
- Durante el tiempo de ejecución, Spring Boot supervisa la carpeta que se encuentra en classpath (en maven, las carpetas que están en la carpeta "target"). Solo necesitamos activar la compilación de las fuentes en los cambios que causarán la actualización de la carpeta 'destino' y Spring Boot reiniciará automáticamente la aplicación. Si estamos utilizando Eclipse IDE, la acción de guardar puede desencadenar la compilación.
- El módulo spring-boot-devtools incluye un servidor LiveReload incorporado que se puede usar para activar una actualización del navegador cuando se cambia un recurso. Las extensiones del navegador LiveReload están disponibles gratuitamente para Chrome, Firefox y Safari desde:

  http://livereload.com/extensions/

  spring.devtools.restart.additional-paths=.

  spring.devtools.livereload.enabled=true

Crear proyecto

- Desde Spring Initializr:
  - https://start.spring.io/
  - Descomprimir en el workspace
  - Import → Maven → Existing Maven Project
- Desde Eclipse:
  - New Project → Sprint Boot → Spring Started Project
- Dependencias
  - Web
  - JPA
  - JDBC (o proyecto personalizado)

pom.xml

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.1.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

- Con las etiquetas <parent></parent> se indica que nuestro POM hereda del de Spring Boot. La dependencia spring-boot-starter-web es la necesaria para poder empezar con un proyecto de tipo MVC, pero a medida que crece la aplicación se iran añadiendo más dependencias.
- Para alterar la configuración dada por defecto se ofrecen campos que se añadirán y asignaran en el archivo application.properties de la carpeta src/main/resources del proyecto.

Application

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication implements CommandLineRunner {

    public static void main(String[] args) {
        SpringApplication.run(ApiHrApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        // Opcional: Procesar los args una vez arrancado SprintBoot

    }

}
```

Configuración

- @Configuration: Indica que esta es una clase usada para configurar el contenedor Spring.
- @ComponentScan: Escanea los paquetes de nuestro proyecto en busca de los componentes que hayamos creado, ellos son, las clases que utilizan las siguientes anotaciones: @Component, @Service, @Controller, @Repository.
- @EnableAutoConfiguration: Habilita la configuración automática, esta herramienta analiza el classpath y el archivo application.properties para configurar nuestra aplicación en base a las librerías y valores de configuración encontrados, por ejemplo: al encontrar el motor de bases de datos H2 la aplicación se configura para utilizar este motor de datos, al encontrar Thymeleaf se crearan los beans necesarios para utilizar este motor de plantillas para generar las vistas de nuestra aplicación web.
- @SpringBootApplication: Es el equivalente a utilizar las anotaciones: @Configuration, @EnableAutoConfiguration y @ComponentScan

Configuración

- Editar src/main/resources/application.properties:

```
# Oracle settings
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=hr
spring.datasource.password=hr
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver

# MySQL settigs
spring.datasource.url=jdbc:mysql://localhost:3306/sakila
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n
logging.level.org.hibernate.SQL=debug

server.port=8080
```

- Repetir con src/test/resources/application.properties

Oracle Driver con Maven

- http://www.oracle.com/technetwork/database/features/jdbc/index-091264.html
- Instalación de Maven:
  - Descargar y descomprimir (https://maven.apache.org)
  - Añadir al PATH: C:\Program Files\apache-maven\bin
  - Comprobar en la consola de comandos: mvn –v
- Descargar el JDBC Driver de Oracle (ojdbc6.jar):
  - https://www.oracle.com/technetwork/apps-tech/jdbc-112010-090769.html
- Instalar el artefacto ojdbc en el repositorio local de Maven
  - mvn install:install-file -Dfile=Path/to/your/ojdbc6.jar -DgroupId=com.oracle -DartifactId=ojdbc6 -Dversion=11.2.0 -Dpackaging=jar
- En el fichero pom.xml:

```
<dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc6</artifactId>
    <version>11.2.0</version>
</dependency>
```


Configuración del proxy: Maven

- Crear fichero setting.xml o editar %MAVEN_ROOT%/conf/setting.xml:

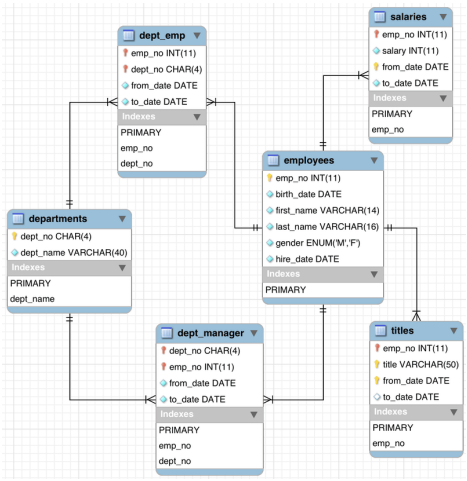
```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">
    <localRepository>C:\directorio\local\.m2\repository</localRepository>
    <proxies>
        <proxy>
        <id>optional</id>
        <active>true</active>
        <protocol>http</protocol>
        <username>usuario</username>
        <password>contraseña</password>
        <host>proxy.dominion.com</host>
        <port>8080</port>
        <nonProxyHosts>local.net|some.host.com</nonProxyHosts>
      </proxy>
    </proxies>
</settings>
```

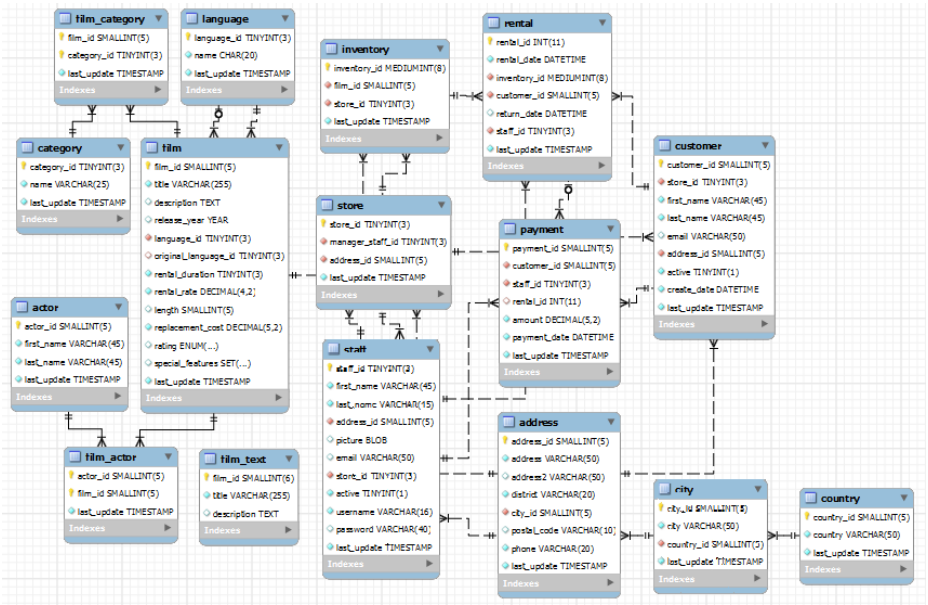
- Referenciarlo en Window → Preferences → Maven → User setting → User setting , browse…, seleccionar fichero recién creado, aceptar, update setting, aplicar y cerrar.

Instalación de MySQL

- Descargar e instalar:
  - https://mariadb.org/download/
- Incluir en la seccion [mysqld] de %MYSQL_ROOT%/data/my.ini
  - default_time_zone='+01:00'
- Descargar bases de datos de ejemplos:
  - https://dev.mysql.com/doc/index-other.html
- Instalar bases de datos de ejemplos:
  - mysql -u root -p < employees.sql
  - mysql -u root -p < sakila-schema.sql
  - mysql -u root -p < sakila-data.sql

Modelos de datos





ARQUITECTURA

Spring Batch

- Muchas aplicaciones dentro del dominio empresarial requieren un procesamiento masivo para realizar operaciones empresariales en entornos de misión crítica. Estas operaciones incluyen:
  - Procesamiento automatizado y complejo de grandes volúmenes de información que se procesa de manera más eficiente sin la interacción del usuario. Estas operaciones suelen incluir eventos basados en el tiempo (como cálculos de fin de mes, avisos o correspondencia).
  - Aplicación periódica de reglas de negocio complejas procesadas repetitivamente en conjuntos de datos muy grandes (por ejemplo, cálculos de beneficios o ajustes de tasa).
  - Integración de la información que se recibe de los sistemas internos y externos que normalmente requieren el formateo, la validación y el procesamiento de manera transaccional en el sistema de registro. El procesamiento por lotes se utiliza para procesar miles de millones de transacciones diarias para las empresas.
- Spring Batch es un marco de trabajo por lotes, ligero y completo, diseñado para permitir el desarrollo de aplicaciones de lotes robustas, vitales para las operaciones diarias de los sistemas empresariales.
- Spring Batch se basa en las características de Spring Framework (productividad, enfoque de desarrollo basado en POJO y facilidad de uso general), al tiempo que facilita a los desarrolladores el acceso y el aprovechamiento de servicios empresariales más avanzados cuando sea necesario.

Escenarios de uso

- Un programa por lotes típico en general:
  - Lee una gran cantidad de registros de una base de datos, archivo o cola.
  - Procesa los datos de alguna manera.
  - Escribe los datos en una forma modificada.
- Spring Batch automatiza esta iteración básica de lotes, brindando la capacidad de procesar transacciones similares como un conjunto, generalmente en un entorno sin conexión, sin la interacción del usuario.
- Los trabajos por lotes forman parte de la mayoría de los proyectos de TI y Spring Batch es un marco de código abierto que proporciona una solución robusta a escala empresarial.

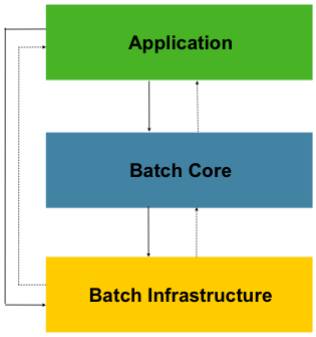
Escenarios de negocios

- Acometer procesos por lotes periódicamente
- Procesamiento por lotes concurrente: procesamiento paralelo de un trabajo
- Procesamiento en serie, dirigido por mensajes de negocio
- Procesamiento por lotes masivos en paralelo
- Reinicio manual o programado después de fallos
- Procesamiento secuencial de pasos dependientes
  - con extensiones batch controlados por flujo de trabajo
- Procesamiento parcial:
  - omitir registros (por ejemplo, en reversión)
- Transacciones de lote completo, para casos con un tamaño de lote pequeño o scripts / procedimientos almacenados existentes

Objetivos

- Utilizar el modelo de programación Spring: concentrarse en la lógica de negocio y dejar que el marco se ocupe de la infraestructura.
- Separación clara de preocupaciones entre la infraestructura, el entorno de ejecución por lotes y la aplicación por lotes.
- Proporcionar servicios comunes de ejecución central como interfaces que todos los proyectos pueden implementar.
- Proporcionar implementaciones simples y predeterminadas de las interfaces de ejecución centrales que están 'listas para usar'.
- Fácil de configurar, personalizar y ampliar los servicios, aprovechando el marco de resorte en todas las capas.
- Todos los servicios básicos existentes deben ser fáciles de reemplazar o extender, sin ningún impacto en la capa de infraestructura.
- Proporcionar un modelo de implementación simple, con los JAR de arquitectura completamente separados de la aplicación, creados con Maven.

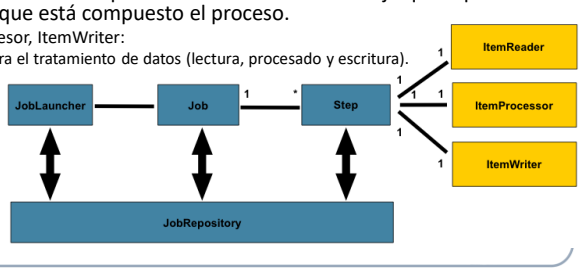
Arquitectura



- Esta arquitectura en capas destaca tres componentes principales de alto nivel: Aplicación, Núcleo e Infraestructura.
- La aplicación contiene todos los trabajos por lotes y el código personalizado escrito por los desarrolladores que utilizan Spring Batch.
- Batch Core contiene las clases en tiempo de ejecución de núcleo necesarias para iniciar y controlar un trabajo por lotes. Incluye implementaciones para JobLauncher, Job y Step.
- Tanto Application como Core están construidos sobre una infraestructura común. Esta infraestructura contiene lectores, escritores y servicios comunes (como el RetryTemplate), que son utilizados tanto por los desarrolladores de aplicaciones (lectores y escritores, como ItemReader y ItemWriter) como el marco central en sí (reintentos, que es su propia biblioteca).

Elementos de un batch

- JobRepository: componente encargado de la persistencia de metadatos relativos a los procesos tales como procesos en curso o estados de las ejecuciones.
- JobLauncher: componente encargado de lanzar los procesos y suministrar los parámetros de entrada deseados.
- Job: El trabajo es la representación del proceso. Un proceso, a su vez, es un contenedor de múltiples pasos (steps), al menos uno.
  - Step: Un paso es un elemento independiente dentro de un trabajo que representa una de las fases de las que está compuesto el proceso.
    - ItemReader, ItemProcesor, ItemWriter:
      - componentes para el tratamiento de datos (lectura, procesado y escritura).



Trabajos

- Job
  - El Job es la representación del proceso. Un proceso, a su vez, es un contenedor de pasos (steps) que combina varios pasos, estableciendo un flujo y permite la configuración de propiedades globales para todos los pasos, como la capacidad de reinicio.
- 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.
- JobInstance
  - Es una representación organizativa de un determinado job con ciertos parámetros de ejecución.
- JobExecution
  - Representa la ejecución correcta o fallida de un determinada instancia de un job en un determinado instante de tiempo. Identifica una ejecución y el estado del job.

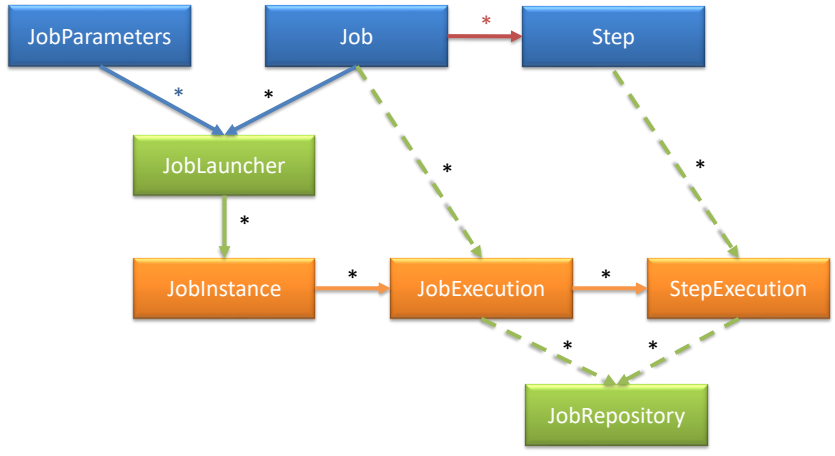
Pasos

- Step
  - Encapsula cada una de las fases o pasos de un trabajo. De este modo un job está compuesto por uno o más Steps. Un Step podrá ser tan simple, complejo o de la tipología que el desarrollador determine oportuno.
- StepExecution
  - Representa cada intento de ejecución de un determinado Step. Cada vez que se ejecuta un Step se creará un nuevo StepExecution.
- Item Readers
  - 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 devolverá null.
- Item Writter:
  - Representa la fase de salida o escritura de información de un Step. 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 procesado de la información. ItemProcesor provee de elementos de transformación de la información entre la fase de lectura (ItemReader) y posterior fase de escritura (ItemWriter). Cuando un ItemProcesor dovuelva null indicará que para dicho elemento no es necesario que se realice la fase de escritura.

Ejecución

- JobLauncher
  - Representa una interfaz simple para lanzar ejecuciones de un Job con un conjunto de JobParameters como entrada.
- 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). En caso se error le permite conocer que parte del proceso se ejecuto.
- 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.

Correlación



CONFIGURACIÓN Y EJECUCIÓN DE UN TRABAJO

Job

- Existen múltiples implementaciones de la interfaz Job, sin embargo, la implementación de los patrones factory y builder permiten abstraerse de las diferencias en la configuración.
- La configuración de un trabajo puede estar basada en Java o conectada a un archivo de configuración XML.
- La configuración del trabajo contiene:
  - Id: El nombre simple del trabajo.
  - Steps: Definición y ordenación de Step.
  - 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 controladores 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.

Configuraciones

- Java:

```
@Bean
public Job footballJob() {
   return this.jobBuilderFactory.get("footballJob")
                .start(playerLoad())
                .next(gameLoad())
                .next(playerSummarization())
                .end()
                .build();
}
```

- XML:

```
<job id="footballJob">
   <step id="playerload" parent="s1" next="gameLoad"/>
   <step id="gameLoad" parent="s2" next="playerSummarization"/>
   <step id="playerSummarization" parent="s3"/>
</job>
```


Reinicio

- El lanzamiento de un Job se considera un 'reinicio' si ya existe el JobExecution del JobInstance. Idealmente, todos los trabajos deberían poder iniciarse donde se quedaron, pero hay escenarios donde esto no es posible. Si un Job nunca se debe reiniciar entonces la propiedad restartable se debe establecer en 'false':

```
return this.jobBuilderFactory.get("footballJob")
                .preventRestart()
```

Interceptar la ejecución del trabajo

- Durante el curso de la ejecución de un trabajo, puede ser útil recibir notificaciones de los eventos del ciclo de vida para ejecutar un código personalizado. La implementación del interfaz lo permite:

```
public interface JobExecutionListener {
    void beforeJob(JobExecution jobExecution);
    void afterJob(JobExecution jobExecution);
}
```

- Para asociarlo:

```
return this.jobBuilderFactory.get("footballJob")
                .listener(sampleListener())
```

- Las anotaciones correspondientes a esta interfaz son:
  - @BeforeJob
  - @AfterJob

Validar parámetros

- Un trabajo puede declarar opcionalmente un validador para los parámetros del trabajo en tiempo de ejecución. Esto es útil cuando, por ejemplo, se necesita comprobar que un trabajo se inicia con todos sus parámetros obligatorios.
- Existe un DefaultJobParametersValidator que se puede usar para restringir combinaciones simples de parámetros obligatorios y opcionales, y para restricciones más complejas puede implementar la interfaz.

```
return this.jobBuilderFactory.get("footballJob")
                .validator(parametersValidator())
```

Herencia de un trabajo padre

- Si un grupo de Trabajos comparte configuraciones similares, pero no idénticas, entonces puede ser útil definir un Job "padre" del cual trabajos concretos puedan heredar propiedades. Al igual que la herencia de clase en Java, el Job "hijo" combinará sus elementos y atributos con los de los padres.

```
<job id="baseJob" abstract="true">
  <listeners>
    <listener ref="listenerOne"/>
  <listeners>
</job>
<job id="job1" parent="baseJob">
  <step id="step1" parent="standaloneStep"/>
  <listeners merge="true">
    <listener ref="listenerTwo"/>
  <listeners>
</job>
```


Configuración Java

- @EnableBatchProcessing proporciona una configuración básica para la creación de trabajos por lotes. Dentro de esta configuración básica, se crea una instancia StepScope y una series de beans disponibles para ser inyectados (autowired):
  - JobRepository - nombre del bean "jobRepository"
  - JobLauncher - nombre de bean "jobLauncher"
  - JobRegistry - nombre de bean "jobRegistry"
  - PlatformTransactionManager - nombre de bean "transactionManager"
  - JobBuilderFactory - nombre de bean "jobBuilders"
  - StepBuilderFactory - nombre del bean "stepBuilders"

Configurar el JobRepository

- El JobRepository permitirá el acceso a la base de datos para almacenar la información relativa a la ejecución del batch, como son los JobExecution y StepExecution, y dotará de métodos a la infraestructura para gestionar el JobLauncher, el Job y los Steps.

```
<job-repository id="jobRepository" data-source="dataSource"
    transaction-manager="transactionManager"
    isolation-level-for-create="SERIALIZABLE"
    table-prefix="BATCH_" 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.batch.core.repository.support.MapJobRepositoryFactoryBean">
    <property name="transactionManager" ref="transactionManager"/>
</bean>
```


Configurar el JobRepository

- No se requiere ninguna de las opciones de configuración enumeradas a continuación, excepto dataSource y transactionManager.

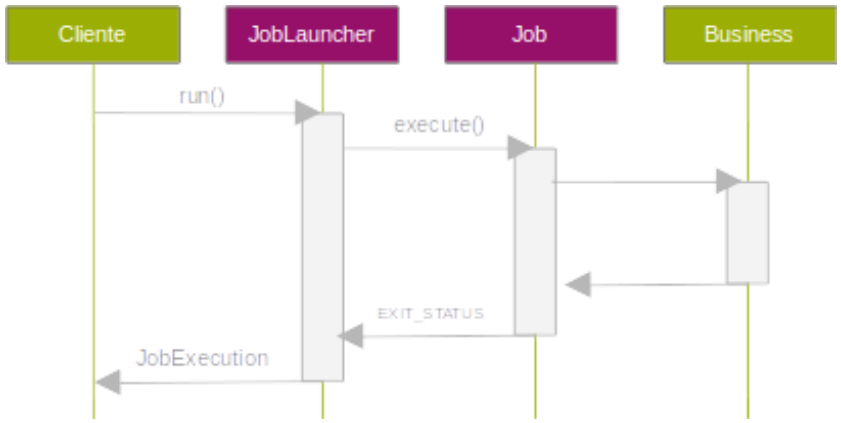
```
@Override
protected JobRepository createJobRepository() throws Exception {
   JobRepositoryFactoryBean factory = new JobRepositoryFactoryBean();
   factory.setDataSource(dataSource);
   factory.setTransactionManager(transactionManager);
   factory.setIsolationLevelForCreate("ISOLATION_SERIALIZABLE");
   factory.setTablePrefix("BATCH_");
   factory.setMaxVarCharLength(1000);
   return factory.getObject();
}
```

Configurar el JobLauncher

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

```
<bean id="jobLauncher" class=
"org.springframework.batch.core.launch.support.SimpleJobLauncher">
   <property name="jobRepository" ref="jobRepository" />
</bean>
@Override
protected JobLauncher createJobLauncher() throws Exception {
      SimpleJobLauncher jobLauncher = new SimpleJobLauncher();
      jobLauncher.setJobRepository(jobRepository);
      jobLauncher.afterPropertiesSet();
      return jobLauncher;
}
```


Configurar el JobLauncher



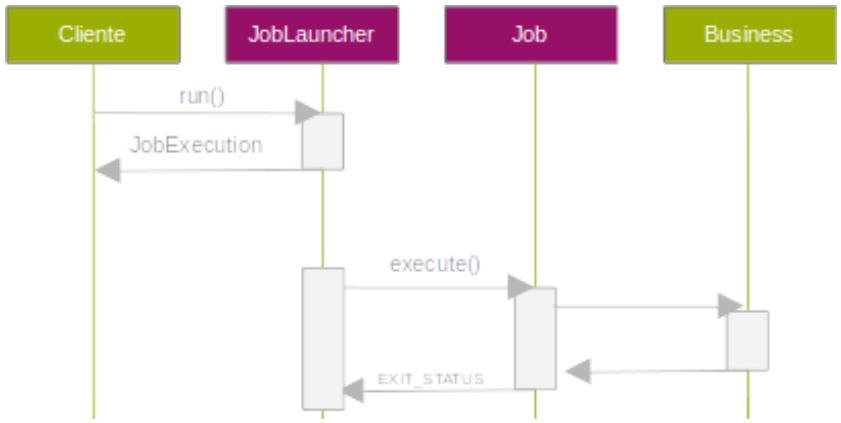
Configurar el JobLauncher

- La secuencia es sencilla y funciona bien cuando se inicia desde un programador pero surgen problemas al intentar iniciar desde una solicitud HTTP, donde el lanzamiento se debe realizar de forma asíncrona.

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

@Bean
public JobLauncher jobLauncher() {
      SimpleJobLauncher jobLauncher = new SimpleJobLauncher();
      jobLauncher.setJobRepository(jobRepository());
      jobLauncher.setTaskExecutor(new SimpleAsyncTaskExecutor());
      jobLauncher.afterPropertiesSet();
      return jobLauncher;
}
```


Configurar el JobLauncher



Ejecución de un Job

- Para poder realizar la ejecución de un Batch se necesitan al menos dos cosas, el JobLauncher y el propio Job 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 Job 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.CommandLineJob
Runner spring/batch/jobs/job-read-files.xml readJob
param1=value1
```


Ejecución de un Job

- Ejecución desde el contexto de ejecución del batch
  - 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.

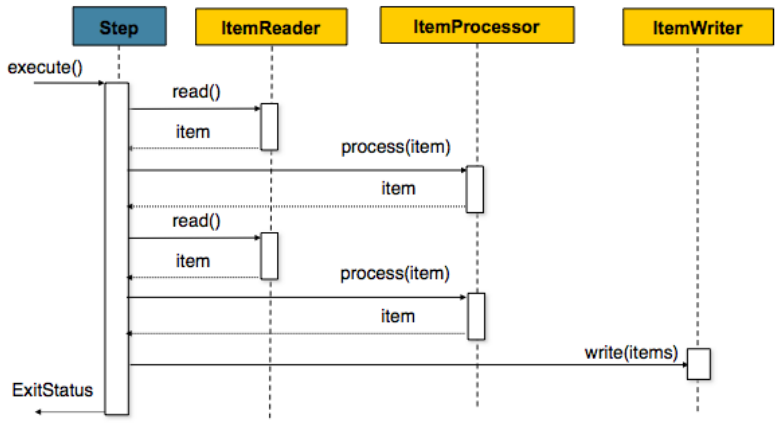
```
@Controller
public class JobLauncherController {
   @Autowired
   JobLauncher jobLauncher;
   @Autowired
   Job job;
   @RequestMapping("/jobLauncher.html")
   public void handle() throws Exception{
      jobLauncher.run(job, new JobParameters());
   }
}
```

CONFIGURACIÓN DE UN PASO

Introducción

- Un Step es un objeto de dominio que encapsula una fase secuencial independiente de un trabajo por lotes y contiene toda la información necesaria para definir y controlar el procesamiento por lotes real.
- Un Step puede ser tan simple como cargar datos de un archivo en la base de datos, requiriendo poco o ningún código, o complejo como reglas de negocio complicadas que se aplican como parte del procesamiento.
- Spring Batch utiliza un estilo de procesamiento "orientado a Chunk" en su implementación más utilizada. El procesamiento orientado a fragmentos se refiere a leer los datos uno por uno y crear "fragmentos" que se escriben dentro de un límite de transacción. Un elemento se lee desde ItemReader, se entrega a ItemProcessor, y se agrega. Una vez que el número de elementos leídos es igual al intervalo de confirmación, el fragmento completo se escribe con el ItemWriter y se confirma la transacción.

Procesamiento orientado a fragmentos



Configurar un Step

- Step es una clase extremadamente compleja que potencialmente puede contener muchos colaboradores.

```
@Bean
public Step sampleStep(PlatformTransactionManager transactionManager) {
    return this.stepBuilderFactory.get("sampleStep")
                .transactionManager(transactionManager)
                .<String, String>chunk(10)
                .reader(itemReader())
                .writer(itemWriter())
                .build();
}
```

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

Configurar un Step

- reader: El ItemReader que proporciona elementos para su procesamiento.
- writer: El ItemWriter que procesa los elementos proporcionados por el ItemReader.
- transactionManager: PlatformTransactionManager que comienza y confirma las transacciones durante el procesamiento.
- repository: El JobRepository que almacena periódicamente los StepExecution y ExecutionContext durante el procesamiento (justo antes de confirmar).
- chunk: Indica que este es un paso basado en un elemento y la cantidad de elementos que se procesarán antes de que se confirme la transacción.
- commit-interval: El número de elementos que se procesarán antes de que se confirme la transacción.

Orientación a tareas

- En determinados casos, un paso no requiere el típico procesamiento lectura/tratamiento/escritura.
- 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.

```
@Bean
public Step deleteFilesInDir() {
    return this.stepBuilderFactory.get("deleteFilesInDir")
                .tasklet(fileDeletingTasklet())
                .build();
}
```

```
<step id="deleteFilesInDir"><tasklet ref="fileDeletingTasklet"/></step>
```

- La interfaz Tasklet contiene únicamente un método execute que será ejecutado repetidamente mientras la devolución del Tasklet sea distinta a RepeatStatus.FINISHED o bien se lance una excepción.

Tasklet

```
public class FileDeletingTasklet implements Tasklet, InitializingBean {
   private Resource directory;
   public RepeatStatus execute(StepContribution contribution, ChunkContext chunkContext) throws Exception {
      File dir = directory.getFile();
      Assert.state(dir.isDirectory());
      File[] files = dir.listFiles();
      for (int i = 0; i < files.length; i++) {
         boolean deleted = files[i].delete();
         if (!deleted) {
            throw new UnexpectedJobExecutionException("Could not delete file " + files[i].getPath());
         }
      }
      return RepeatStatus.FINISHED;
   }
   public void setDirectoryResource(Resource directory) {
      this.directory = directory;
   }
   public void afterPropertiesSet() throws Exception {
      Assert.notNull(directory, "directory must be set");
   }
}
```


TaskletAdapter

- Al igual que con otros adaptadores para las interfaces ItemReader y ItemWriter, la interfaz Tasklet contiene una implementación que permite adaptarse a cualquier clase preexistente: TaskletAdapter.

```
@Bean
public MethodInvokingTaskletAdapter myTasklet() {
   MethodInvokingTaskletAdapter adapter = new MethodInvokingTaskletAdapter();
   adapter.setTargetObject(fooDao());
   adapter.setTargetMethod("updateFoo");
   return adapter;
}
<bean id="myTasklet" class="...MethodInvokingTaskletAdapter">
   <property name="targetObject">
      <bean class="org.mycompany.FooDao"/>
   </property>
   <property name="targetMethod" value="updateFoo" />
</bean>
```

Control de flujo

- Cuando un trabajo cuenta con mas de un paso existe la necesidad de poder controlar cómo el trabajo "fluye" de un paso a otro: el fallo un Step no significa necesariamente que el Job deba fallar o puede haber más de un tipo de "éxito" al realizarse un Step que determina que Step debe ejecutarse a continuación.
- El escenario de flujo más simple es un trabajo en el que todos los pasos se ejecutan secuencialmente y, si un paso falla, falla todo el trabajo:

```
@Bean
public Job job() {
      return this.jobBuilderFactory.get("job")
                        .start(stepA())
                        .next(stepB())
                        .next(stepC())
                        .build();
}
<job id="job">
   <step id="stepA" parent="s1" next="stepB" />
   <step id="stepB" parent="s2" next="stepC"/>
   <step id="stepC" parent="s3" />
</job>
```


Flujo condicional

- El método on() utiliza un esquema de coincidencia de patrón simple de coincidencia con el ExitStatus resultante de la ejecución de un Step para de decidir la siguiente acción a realizar.
- Solo se permiten dos caracteres especiales en el patrón:
  - "*" coincide con cero o más caracteres
  - "?" coincide exactamente con un caracter
- El método from() permite recuperar el ExitStatus de un Step cuando la condición es múltiple.
- El método to() permite indicar el Step a ejecutar en caso de que se cumpla la condición (next condicional).
- Los métodos end(), stopAndRestart() y fail() detienen el trabajo con BatchStatus y ExitStatus del Job como COMPLETED, STOPPED o FAILED respectivamente.
  - El método end tiene un parámetro opcional para personalizar el ExitStatus.
  - La detención con stopAndRestart proporciona una interrupción temporal en el procesamiento, de modo que el operador pueda realizar alguna acción antes de reiniciar Job.

Flujo condicional

```
@Bean
public Job job() {
      return this.jobBuilderFactory.get("job")
                  .start(step1()).on("FAILED").fail()
                  .from(step1()).on("COMPLETED WITH SKIPS").to(errorPrint1())
                  .from(step1()).on("*").to(step2())
                  .end()
                  .build();
}
```

```
<step id="step1" parent="s1">
   <fail on="FAILED" />
   <next on="COMPLETED WITH SKIPS" to="errorPrint1" />
   <next on="*" to="step2" />
</step>
```


MECANISMOS DE LECTURA Y ESCRITURA

ItemReader

- ItemReader es el medio para proporcionar datos de muchos tipos diferentes de entrada. Los casos más comunes:
  - Archivo plano: los lectores de elementos de archivo plano leen líneas de datos de un archivo plano que normalmente describe registros con campos de datos definidos por posiciones fijas en el archivo o delimitados por algún carácter especial (como una coma).
  - XML: los lectores de XML procesan XML independientemente de las tecnologías utilizadas para analizar, asignar y validar objetos. Los datos de entrada permiten la validación de un archivo XML contra un esquema XSD.
  - Base de datos: se accede a un recurso de base de datos para devolver los conjuntos de resultados que se pueden asignar a objetos para su procesamiento. Los lectores de SQL invocan a RowMapper para devolver objetos, realizar un seguimiento de la fila actual si es necesario reiniciar, almacenar estadísticas básicas y proporcionar algunas mejoras de transacción que se explican más adelante.

ItemReader

- ItemReader es un interfaz básico para operaciones de entrada genéricas:

```
public interface ItemReader<T> {
   T read() throws Exception, UnexpectedInputException, ParseException,
        NonTransientResourceException;
}
```

- El método read define el contrato esencial del ItemReader: devuelve un elemento o null si no quedan más elementos. El elemento puede representar una línea en un archivo, una fila en base de datos o un nodo en un archivo XML. En general, se espera que éstos se asignan a un objeto de dominio utilizable, pero no es obligatorio.
- Se espera que las implementaciones de la interfaz sean de solo avance. Sin embargo, si el recurso subyacente es transaccional (como una cola JMS), la llamada read puede devolver el mismo elemento lógico en las llamadas subsiguientes en un escenario de reversión.
- La falta de elementos para procesar por parte de un objeto ItemReader no genera una excepción: en base de datos una consulta que selecciona 0 resultados devuelve null en la primera invocación de lectura.

ItemWriter

- ItemWriter es similar en funcionalidad a un ItemReader pero con operaciones inversas. Los recursos también deben ubicarse, abrirse y cerrarse, pero difieren en que se ItemWriter escribe, en vez de leer. En el caso de bases de datos o colas, estas operaciones pueden ser inserciones, actualizaciones o envíos. El formato de la serialización de la salida es específico para cada trabajo por lotes.
- Al igual que ItemReader, ItemWriteres una interfaz bastante genérica:

```
public interface ItemWriter<T> {
    void write(List<? extends T> items) throws Exception;
}
```

- El método write proporciona el contrato básico de ItemWriter: intenta escribir la lista de elementos pasados mientras esté abierto.
- Debido a que generalmente se espera que los elementos se 'agrupen' en una porción y luego se envíen, la interfaz acepta una lista de elementos, en lugar de un único elemento.
- Después de escribir la lista, cualquier descarga necesaria se puede realizar antes de regresar del método de escritura: si se escribe en un DAO de Hibernate, se pueden hacer varias llamadas para escribir, una para cada elemento y el escritor puede invocar al flush de la sesión Hibernate antes de regresar.

ItemProcessor

- En muchos casos el elemento leído no se puede escribir directamente, hay que "transformar" el elemento antes de que se escriba. Para estos escenarios, Spring Batch proporciona la interfaz ItemProcessor:

```
public interface ItemProcessor<I, O> {
    O process(I item) throws Exception;
}
```

- Cuenta con un simple método process que recibe un objeto, lo transforma y devuelve otro. El objeto proporcionado puede o no ser del mismo tipo. La lógica de negocios de la transformación se aplica dentro del proceso. Si el método devuelve null no se agrega a la lista de elementos entregados a ItemWriter: filtrado de elementos.
- El ItemProcessor se conecta directamente a un paso y debe ser único, pero se puede utilizar un patrón Composite con CompositeItemProcessor:

```
CompositeItemProcessor<Foo,Foobar> compositeProcessor =
                    new CompositeItemProcessor<Foo,Foobar>();
List itemProcessors = new ArrayList();
itemProcessors.add(new FooTransformer());
itemProcessors.add(new BarTransformer());
compositeProcessor.setDelegates(itemProcessors);
```

```
<bean id="compositeItemProcessor"
      class="...CompositeItemProcessor">
  <property name="delegates">
    <list>
      <bean class="..FooProcessor" />
      <bean class="..BarProcessor" />
    </list>
  </property>
</bean>
```

ItemStream

- En general, como parte del alcance de un trabajo por lotes, los lectores y escritores deben abrirse, cerrarse y requieren un mecanismo para mantener el estado.
- La interfaz ItemStream cumple con este propósito:

```
public interface ItemStream {
    void open(ExecutionContext executionContext) throws ItemStreamException;
    void update(ExecutionContext executionContext) throws ItemStreamException;
    void close() throws ItemStreamException;
}
```

- Los clientes ItemReader e ItemWriter que también implementen ItemStream deben llamar al método open antes de cualquier invocación a read/write, para abrir el recurso, archivo u obtener conexiones.
- A la inversa, el método close se invoca para garantizar que los recursos asignados durante la apertura se liberen de forma segura.
- El método update se invoca para garantizar que cualquier estado que se mantiene actualmente se carga en el ExecutionContext: se invoca antes de confirmar, para garantizar que el estado actual se mantenga en la base de datos antes de confirmar.

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.example.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.

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 apoyá 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

- 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:
  - 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.

```
<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>
```


Database ItemWriters

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

```
<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.example.....CustomItemSetter" />
    </property>
</bean>
```

StaxEventItemReader

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

```
<bean id="itemReader" class="org.springframework.batch.item.xml.StaxEventItemReader">
  <property name="fragmentRootElementName" value="trade" />
  <property name="resource" value="data/iosample/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.oxm.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

- Para realizar la escritura de información en XML a través del StAX API.

```
<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.

```
<bean id="customerCreditMarshaller"
    class="org.springframework.oxm.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>
```

- overwriteOutput: Sobreescribe el fichero de salida en caso de existir.

Otros

- Decoradores
  - SynchronizedItemStreamReader
  - SingleItemPeekableItemReader
  - MultiResourceItemWriter
  - ClassifierCompositeItemWriter
  - ClassifierCompositeItemProcessor
- Lectores y escritores de mensajes
  - AmqpItemReader
  - AmqpItemWriter
  - JmsItemReader
  - JmsItemWriter
  - SimpleMailMessageItemWriter
- Lectores LDAP
  - LdifReader
  - MappingLdifReader
- Lectores de bases de datos
  - Neo4jItemReader
  - MongoItemReader
  - HibernateCursorItemReader
  - HibernatePagingItemReader
  - RepositoryItemReader
- Escritores de bases de datos
  - Neo4jItemWriter
  - MongoItemWriter
  - RepositoryItemWriter
  - HibernateItemWriter
  - JdbcBatchItemWriter
  - JpaItemWriter
  - GemfireItemWriter

ESCALABILIDAD Y CONCURRENCIA

Introducción

- Muchos problemas de procesamiento por lotes se pueden resolver con trabajos de proceso único y de un solo subproceso, por lo que siempre es una buena idea verificar si esto satisface las necesidades antes de pensar en implementaciones más complejas.
- Hay que hacer una medición realista del desempeño de un trabajo real y ver si la implementación más simple satisface la necesidades primero.
- Se puede leer y escribir un archivo de varios cientos de megabytes en menos de un minuto, incluso con hardware estándar.
- Cuando sea necesario implementar un trabajo con un procesamiento paralelo, Spring Batch ofrece una amplia gama de opciones.
- A primer nivel hay dos modos de procesamiento paralelo:
  - Proceso único, multihilo
  - Multiproceso
- Estos se desglosan en categorías también, de la siguiente manera:
  - Paso multi-hilo (proceso único)
  - Pasos paralelos (proceso único)
  - Chunking remoto de paso (multiproceso)
  - Partición de un paso (proceso simple o múltiple)

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 procesado de elementos de manera no consecutiva. En algunas situaciones será necesario limitar el número de hilos, para ello se especificará el parámetro throttle-limit.

Steps paralelos

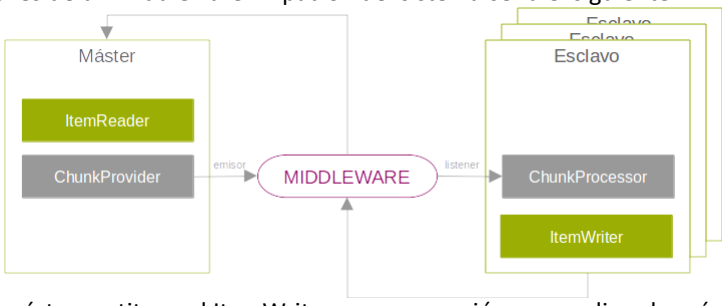
- En la definición de la estructura de determinados batchs es posible identificar cierta lógica u operativa que es necesaria paralelizar. Para ello es posible particionar y delegar responsabilidades de la operativa asignándoles steps individuales que poder paralelizar en un único proceso. La configuración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

Remote chunking

- La técnica denominada Remote chunking consiste en derivar el procesado del step a través de múltiples procesos remotos comunicados entre sí a través de un middleware. El patrón del sistema sería el siguiente:



- El máster sustituye el ItemWriter por una versión que realiza el envío de elementos al middleware, mientras que los esclavos sustituyen el ItemReader por listeners al middleware para procesar los elementos.

GESTIÓN DE ERRORES

Eventos de error

- Una parte orientada Step (creada a partir de los StepBuilderFactory) permite implementar este caso de uso con un simple ItemReadListener para errores read y ItemWriteListener para errores write.
- El siguiente fragmento de código ilustra una escucha que registra los errores de lectura y escritura:

```
public class ItemFailureLoggerListener extends ItemListenerSupport {
   private static Log logger = LogFactory.getLog("item.error");
   public void onReadError(Exception ex) {
      logger.error("Encountered error on read", e);
   }
   public void onWriteError(Exception ex, List<? extends Object> items) {
      logger.error("Encountered error on write", ex);
   }
}
```

- Una vez implementado este escucha, se debe registrar con un paso, como se muestra en el siguiente ejemplo:

```
@Bean
public Step simpleStep() {
    return this.stepBuilderFactory.get("simpleStep")
                    ...
                    .listener(new ItemFailureLoggerListener())
                    .build();
```

Skip

- Hay muchos escenarios en los que los errores encontrados durante el procesamiento no deberían dar como resultado un Step fallido, sino que deberían omitirse. Esta suele ser una decisión que debe ser tomada por alguien que entiende los datos en sí y qué significado tiene. Es posible que los datos financieros, por ejemplo, no se puedan omitir porque dan como resultado la transferencia de dinero, lo que debe ser completamente exacto. Cargar una lista de proveedores, por otro lado, podría permitir saltos. Si un proveedor no se cargue porque el formato no fue correcto o faltaba la información necesaria, es probable que no haya problemas. Por lo general, estos registros erróneos también se registran, mediante los controladores de eventos.
- El siguiente ejemplo muestra un ejemplo del uso de un límite de omisión:

```
@Bean
public Step step1() {
    return this.stepBuilderFactory.get("step1")
                .<String, String>chunk(10)
                .reader(flatFileItemReader())
                .writer(itemWriter())
                .faultTolerant()
                .skipLimit(10)
                .skip(FlatFileParseException.class)
                .build();
}
```

- El uso de .skip(Exception.class) provoca saltos en todas las excepciones y es puede matizar con .noSkip(FileNotFoundException.class).

Reintentos

- En la mayoría de los casos, se desea que una excepción cause un salto o un fallo del Step. Sin embargo, no todas las excepciones son deterministas. Si se produce una FlatFileParseException mientras se lee, siempre se lanza para ese registro dado que restablecer el ItemReader no ayuda.
- Sin embargo, para otras excepciones, como un DeadlockLoserDataAccessException, que indica que el proceso actual ha intentado actualizar un registro en el que otro proceso tiene un bloqueo activado, esperar e intentarlo de nuevo puede resultar útil.
- En este caso, el reintento se debe configurar de la siguiente manera:

```
@Bean
public Step step1() {
    return this.stepBuilderFactory.get("step1")
                .<String, String>chunk(2)
                .reader(itemReader())
                .writer(itemWriter())
                .faultTolerant()
                .retryLimit(3)
                .retry(DeadlockLoserDataAccessException.class)
                .build();
}
```

- El Step permite un límite para el número de veces que un elemento individual puede ser reintentado de nuevo y una lista de excepciones que son 'reintentable'.

CASO PRACTICO

Inicio

- Proyecto
  - Dependencias: Batch, HSQLDB
- Generadores de CSV:
  - https://mockaroo.com/
  - http://www.generatedata.com/?lang=es
- Generar fichero personas-1.csv de entrada:
  - id,first_name,last_name,email,gender,ip_address
- Configuración de la base de datos (src/main/resources/schema-all.sql):

```
DROP TABLE personas IF EXISTS;
CREATE TABLE personas  (
    id BIGINT IDENTITY NOT NULL PRIMARY KEY,
    nombre VARCHAR(250),
    correo VARCHAR(250),
    ip VARCHAR(20)
);
```

Creación de DTOs y entidades

- package com.example.demo.model;
- Entidad:

```
public class Persona {
    private long id;
    private String nombre, correo, ip;
    // Ctor(), Get/Set, toString(), …
}
```

- DTO:

```
public class PersonaDTO {
    private long id;
    private String nombre, apellidos, correo, sexo, ip;
    // Ctor(), Get/Set, toString()
}
```


ItemProcessor

```
package com.example.demo.batch;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.item.ItemProcessor;
import org.springframework.stereotype.Component;

import com.example.demo.model.Persona;
import com.example.demo.model.PersonaDTO;

@Component
public class PersonaItemProcessor implements ItemProcessor<PersonaDTO, Persona> {
    private static final Logger log = LoggerFactory.getLogger(PersonaItemProcessor.class);

    @Override
    public Persona process(PersonaDTO item) throws Exception {
        if(item.getId() % 2 == 0 || "Male".equals(item.getSexo())) return null;
        Persona rslt = new Persona(item.getId(), item.getApellidos() + ", " + item.getNombre(),
                item.getCorreo(), item.getIp());
        log.info("Procesando: " + item);
        return rslt;
    }
}
```

PersonasJobListener

```
package com.example.demo.batch;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.core.BatchStatus;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.listener.JobExecutionListenerSupport;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

import com.example.demo.model.Persona;

@Component
public class PersonasJobListener extends JobExecutionListenerSupport {
	private static final Logger log = LoggerFactory.getLogger(PersonasJobListener.class);
	@Autowired
	private JdbcTemplate jdbcTemplate;

	@Override
	public void afterJob(JobExecution jobExecution) {
		if(jobExecution.getStatus() == BatchStatus.COMPLETED) {
			log.info("----------------> Finalizado");
			jdbcTemplate.query("SELECT id, nombre, correo, ip FROM personas",
					(rs, row) -> new Persona(rs.getLong(1), rs.getString(2), rs.getString(3), rs.getString(4)))
			.forEach(p -> log.info("Fila: " + p));
		}
	}
}
```


PersonasBatchConfiguration

```
package com.example.demo.batch;
import javax.sql.DataSource;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.core.launch.support.RunIdIncrementer;
import org.springframework.batch.item.database.BeanPropertyItemSqlParameterSourceProvider;
import org.springframework.batch.item.database.JdbcBatchItemWriter;
import org.springframework.batch.item.database.JdbcCursorItemReader;
import org.springframework.batch.item.database.builder.JdbcBatchItemWriterBuilder;
import org.springframework.batch.item.database.builder.JdbcCursorItemReaderBuilder;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.FlatFileItemWriter;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.batch.item.file.builder.FlatFileItemWriterBuilder;
import org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper;
import org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor;
import org.springframework.batch.item.file.transform.DelimitedLineAggregator;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.FileSystemResource;
import org.springframework.jdbc.core.BeanPropertyRowMapper;

import com.example.demo.model.Persona;
import com.example.demo.model.PersonaDTO;

@Configuration
@EnableBatchProcessing
public class PersonasBatchConfiguration {
	@Autowired
	public JobBuilderFactory jobBuilderFactory;
	@Autowired
	public StepBuilderFactory stepBuilderFactory;
```

CSV a DB

```
public FlatFileItemReader<PersonaDTO> personaCSVItemReader(String fname) {
    return new FlatFileItemReaderBuilder<PersonaDTO>().name("personaCSVItemReader")
            .resource(new ClassPathResource(fname))
            .linesToSkip(1)
            .delimited()
            .names(new String[] { "id", "nombre", "apellidos", "correo", "sexo", "ip" })
            .fieldSetMapper(new BeanWrapperFieldSetMapper<PersonaDTO>() { {
                setTargetType(PersonaDTO.class);
            }})
            .build();
}

@Autowired
public PersonaItemProcessor personaItemProcessor;

@Bean
public JdbcBatchItemWriter<Persona> personaDBItemWriter(DataSource dataSource) {
    return new JdbcBatchItemWriterBuilder<Persona>()
            .itemSqlParameterSourceProvider(new BeanPropertyItemSqlParameterSourceProvider<>())
            .sql("INSERT INTO personas VALUES (:id,:nombre,:correo,:ip)")
            .dataSource(dataSource)
            .build();
}
```


CSV a DB

```
@Bean
public Step importCSV2DBStep1(JdbcBatchItemWriter<Persona> personaDBItemWriter) {
    return stepBuilderFactory.get("importCSV2DBStep1")
            .<PersonaDTO, Persona>chunk(10)
            .reader(personaCSVItemReader("personas-1.csv"))
            .processor(personaItemProcessor)
            .writer(personaDBItemWriter)
            .build();
}

@Bean
public Job personasJob(PersonasJobListener listener, Step importCSV2DBStep1) {
    return jobBuilderFactory
            .get("personasJob")
            .incrementer(new RunIdIncrementer())
            .listener(listener)
            .start(importCSV2DBStep1)
            .build();
}
```

DB a CSV

```
@Bean
JdbcCursorItemReader<Persona> personaDBItemReader(DataSource dataSource) {
	return new JdbcCursorItemReaderBuilder<Persona>().name("personaDBItemReader")
			.sql("SELECT id, nombre, correo, ip FROM personas").dataSource(dataSource)
			.rowMapper(new BeanPropertyRowMapper<>(Persona.class))
			.build();
}

@Bean
public FlatFileItemWriter<Persona> personaCSVItemWriter() {
	return new FlatFileItemWriterBuilder<Persona>().name("personaCSVItemWriter")
			.resource(new FileSystemResource("output/outputData.csv"))
			.lineAggregator(new DelimitedLineAggregator<Persona>() {
				{
					setDelimiter(",");
					setFieldExtractor(new BeanWrapperFieldExtractor<Persona>() {
						{
							setNames(new String[] { "id", "nombre", "correo", "ip" });
						}
					});
				}
			}).build();
}
```


DB a CSV

```
@Bean
public Step exportDB2CSVStep(JdbcCursorItemReader<Persona> personaDBItemReader) {
	return stepBuilderFactory.get("exportDB2CSVStep")
			.<Persona, Persona>chunk(100)
			.reader(personaDBItemReader)
			.writer(personaCSVItemWriter())
			.build();
}

@Bean
public Job personasJob(PersonasJobListener listener, Step importCSV2DBStep1,
		Step exportDB2CSVStep) {
	return jobBuilderFactory.get("personasJob")
			.incrementer(new RunIdIncrementer())
			.listener(listener)
			.start(importCSV2DBStep1)
			.next(exportDB2CSVStep)
			.build();
}
```