



# Spring batch

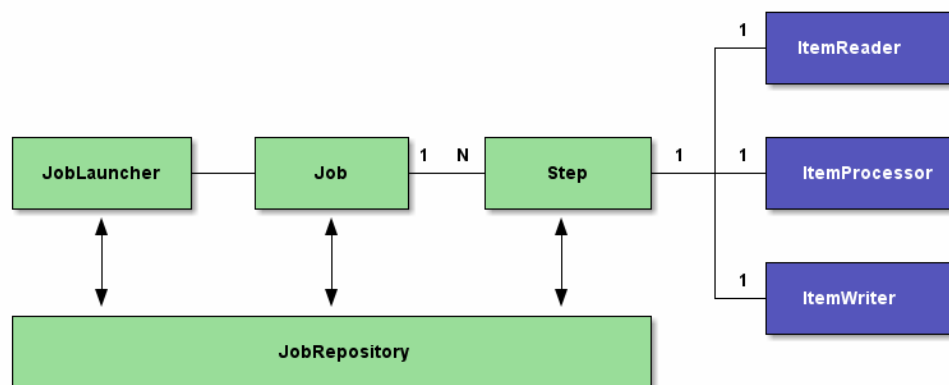
**Spring Batch** es un framework enfocado en el desarrollo de aplicaciones de procesamiento por lotes. Este framework es un módulo de **Spring** y fue desarrollado en colaboración con SpringSource y Accenture.

Los procesos por lotes suelen ser procesos que tratan una gran cantidad de información y generalmente de manera programada sin necesidad de intervención humana. Algunas de sus funcionalidades son:

- **Procesamiento por pasos** → Divide el trabajo en unidades de procesamiento más pequeñas llamadas pasos.
- **Manejo de transacciones** → Garantiza que los datos se procesen correctamente incluso en caso de fallos.
- **Control de reintentos y saltos** → Permite definir cómo manejar errores y reintentar o saltar pasos fallidos.
- **Gestión de estados** → Mantiene el estado en cada ejecución de lote para soportar la reanudación en caso de interrupciones.
- **Escalabilidad** → Soporta la ejecución de trabajos en paralelo y en múltiples nodos.

## Componentes de Spring Batch

Spring Batch propone un diseño modular y flexible para el procesamiento por lotes.



## Componentes clave

- **Job**
  - Tiene un nombre único y puede tener múltiples instancias de ejecución.
  - Es una entidad que representa el trabajo por lotes.
  - Es en esta entidad donde se define la lógica del procesamiento por lotes.
  - Un **Job** puede estar compuesto por varios pasos **Steps**.
  - **JobInstance** → Es una ejecución específica de un **Job**. Cada vez que un **Job** se ejecuta, se crea una nueva **JobInstance**.
  - **JobExecution** → Es una ejecución específica de una **JobInstance** que contiene información sobre la ejecución actual, como el estado.

- **Step**

- Un **Step** es una parte de un **Job** y representa una unidad individual de trabajo que se ejecuta en una transacción.
- Cada **Step** tiene configuración propia y puede ser independiente o depender de otro **Step**.
- **StepExecution** → Representa la ejecución de un **Step** específico dentro de un **Job**.

## Componentes Funcionales

- **ItemReader** → Se utiliza para leer los datos de una fuente de entrada. Puede ser una base de datos, un archivo, cola de mensajes, etc.
- **ItemProcessor** → Se utiliza para procesar y transformar los datos leídos. Es posible aplicar lógica de negocio, validaciones, filtrado, etc.
- **ItemWriter** → Se utiliza para escribir los datos procesados a un destino, el cual puede ser una base de datos, archivo, etc.

## Infraestructura

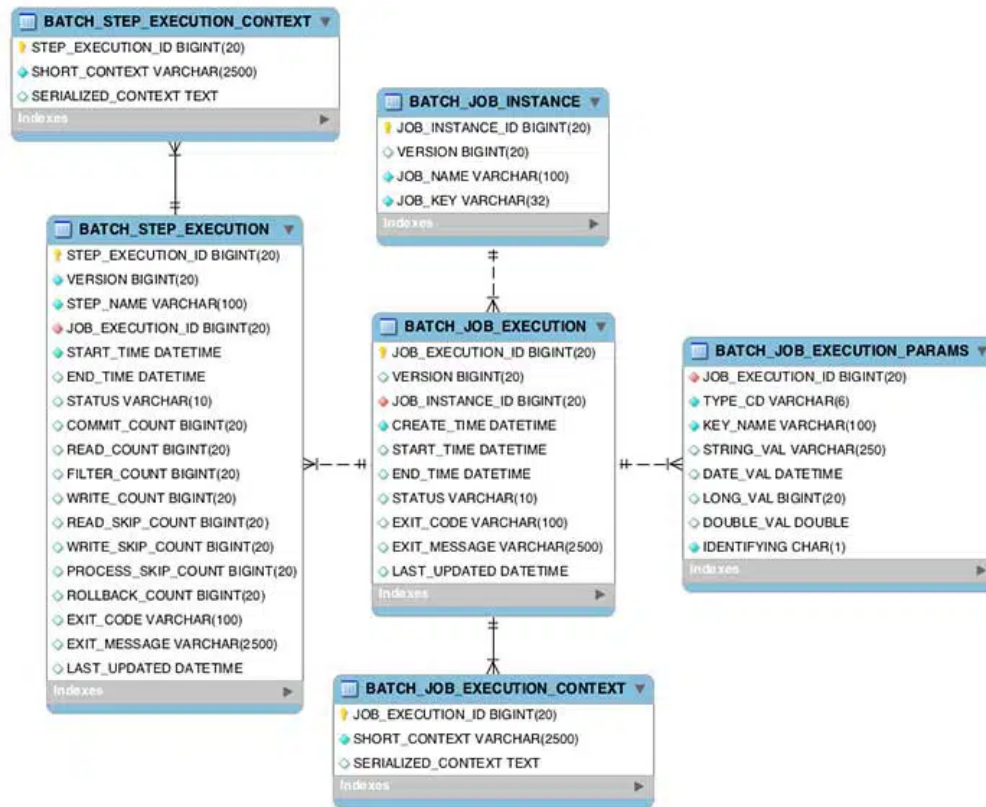
- **Job Repository** → Tiene como tarea almacenar el estado de los **Jobs**, **Steps** y sus respectivas ejecuciones. Esto permite la persistencia y recuperación del estado de las ejecuciones. La base de datos de este repositorio debe ser transaccional.
- **Transaction Management** → Gestiona las transacciones para asegurar que los **Steps** se ejecuten de manera atómica. Si un **Step** falla, la transacción puede ser revertida.
- **JobLauncher** → Se encarga de ejecutar los **Jobs**. Puede ser configurado para ejecutarlos de manera secuencial o concurrente.

## Flujo de ejecución

1. Configuración del **Job** → Definición de los **Jobs** y sus respectivos **Steps** a través de configuraciones XML o anotaciones en Java.
2. Ejecución del **Job** → El **JobLauncher** inicia el **Job**, creando una nueva **JobInstance** y **JobExecution**.
3. Ejecución de los **Steps** → Cada **Step** se ejecuta secuencialmente o en paralelo, según la configuración, leyendo datos con **ItemReader**, procesándolos mediante **ItemProcessor** y escribiéndolos con **ItemWriter**.
4. Persistencia y monitoreo → El estado de cada **Job** y **Step** se almacena en **JobRepository**, permitiendo la monitorización, reinicio y gestión de fallos.

## Esquema de meta datos

Se refiere a la estructura de base de datos utilizada para almacenar la información sobre las ejecuciones de los **Jobs** y **Steps**. El esquema permite rastrear el estado y el historial de los **Jobs** ejecutados, proporcionando características esenciales como la capacidad de reanudar los **Jobs** fallidos, monitorear procesos y generar informes de ejecución.



- BATCH\_JOB\_INSTANCE → Tabla destinada a almacenar la información referente a las instancias de **Job**.
- BATCH\_JOB\_EXECUTION → Almacena la información de las ejecuciones de los **Jobs**. Permite conocer el estado y parámetros relacionados a la ejecución.
- BATCH\_JOB\_EXECUTION\_PARAMS → Almacena los parámetros utilizados durante la ejecución de un **Job**.
- BATCH\_JOB\_EXECUTION\_CONTEXT → Permite almacenar el contexto de la ejecución de los **Jobs**.
- BATCH\_STEP\_EXECUTION → Almacena información sobre las ejecuciones de los **Steps**.
- BATCH\_STEP\_EXECUTION\_CONTEXT → Permite almacenar el contexto de la ejecución de los **Steps**.

## Configuración a nivel de Step

### Chunks

Es una unidad de procesamiento en el que los datos se leen, procesan y escriben en fragmentos manejables. Esta técnica es altamente empleada para el manejo de grandes datos de manera eficiente. Los pasos del procesamiento en chunks pueden resumirse como:

- Lectura → Los datos se leen, a través de un **ItemReader**, en fragmentos hasta que se alcanza el tamaño de chunk predefinido.
- Procesamiento → Cada fragmento es procesado por un **ItemProcessor**.
- Escritura → Después de que un chunk completo ha sido leído y procesado, los elementos se escriben en un destino específico utilizando **ItemWriter**.

### Tasklets

Es un objeto que contiene lógica para ser ejecutada como parte de un *Job*. Se construye con la implementación de la interfaz Tasklets y es la forma más básica para ejecutar un código. Mientras que los chunks están diseñados para el procesamiento masivo, los tasklets permiten realizar tareas más específicas, flexibles y diversas, esto los hace útiles para tareas que no encajen en el modelo de lectura-procesamiento y escritura.

## Flujo de Steps

Permite la configuración y ejecución de estos *Steps* de una manera lógica y ordenada. El flujo se puede controlar de varias maneras:

1. Lineal → Cada *Step* se ejecuta uno tras otro en el orden definido.
2. Condicional → La ejecución depende del resultado de un *Step* anterior.
3. Paralelo → Varios *Steps* se ejecutan en paralelo para aprovechar los recursos del sistema.

## Referencias

### Overview :: Spring Batch


Spring Batch

architecture, general batch principles, batch processing strategies.

 <https://docs.spring.io/spring-batch/reference/>

### An Introduction to Spring Batch - DZone


In this post, we look at how to use Spring Batch and the Quartz Scheduler to run large amounts of data on your applications with job-processing statistics.

 <https://dzone.com/articles/spring-batch>



### GitHub - maldiny/Spring-Batch-en-Castellano: Ejemplos prácticos de Spring Batch

Ejemplos prácticos de Spring Batch. Contribute to maldiny/Spring-Batch-en-Castellano development by creating an account on GitHub.

 <https://github.com/maldiny/Spring-Batch-en-Castellano>

### maldiny/Spring-Batch-en-Castellano

Ejemplos prácticos de Spring Batch



1 Contributor 0 Issues 16 Stars 13 Forks