

# Automatización de Procesamiento de Datos con Spring Batch

## Introducción

En este proyecto desarrollamos una solución automatizada para el procesamiento y carga de datos provenientes de un archivo CSV utilizando Spring Batch. El objetivo principal fue leer información estructurada desde un archivo plano y almacenarla en una base de datos MySQL de forma eficiente y controlada.

Para garantizar un entorno consistente y fácilmente replicable, se utilizó Docker para levantar tanto el contenedor de la base de datos como el entorno de ejecución de la aplicación. Esto permitió simplificar la configuración inicial, facilitar las pruebas y asegurar la portabilidad del sistema.

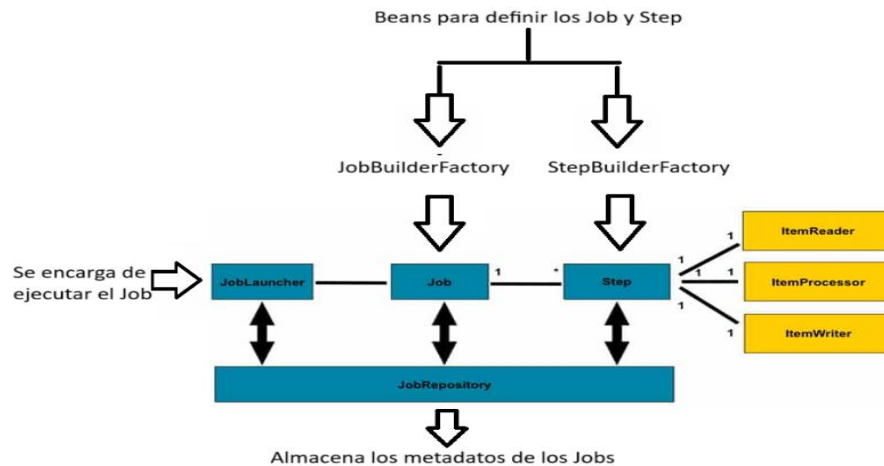
## Tecnologías utilizadas:

- **Spring Batch:** para definir y ejecutar el flujo de procesamiento por etapas (lectura, transformación y escritura).
- **MySQL:** como base de datos relacional para almacenar los registros procesados.
- **Docker:** para contenerizar tanto la base de datos como la aplicación y permitir un despliegue más flexible.
- **CSV:** como formato de entrada de los datos.

## Funcionamiento general:

1. Al iniciar el job de Spring Batch, se lee línea por línea el archivo CSV.
2. Cada línea se convierte en un objeto Java (ItemReader).
3. Se puede aplicar una lógica de validación o transformación (ItemProcessor).
4. Los datos se insertan en la base de datos (ItemWriter).

Este enfoque permite procesar grandes volúmenes de datos sin sobrecargar la memoria y con mecanismos de control de errores y reinicio en caso de fallas



La imagen representa la arquitectura de Spring Batch. Todo comienza con los beans `JobBuilderFactory` y `StepBuilderFactory`, los cuales se utilizan para definir y construir los jobs y steps. Un job es una unidad de trabajo compuesta por uno o varios steps, y cada step se encarga de una fase específica del procesamiento de datos.

Para ejecutar un job, se utiliza el `JobLauncher`, que inicia la ejecución y se comunica con el `JobRepository`, encargado de almacenar los metadatos y el historial de ejecuciones. Cada step está conformado por tres componentes clave:

- `ItemReader`, que lee los datos desde una fuente,
- `ItemProcessor`, que transforma o procesa los datos, y
- `ItemWriter`, que guarda los datos procesados en el destino final.

## Configuración del Proyecto

La configuración del proyecto se hizo utilizando Spring Boot con soporte para Spring Batch. Se definieron las propiedades necesarias para establecer la conexión con la base de datos MySQL y también la ruta del archivo CSV que se va a procesar.

Las configuraciones se declararon en el archivo `application.properties`.

Por ejemplo:

```
target > classes > application.properties
1 # Especifica el nombre del driver JDBC que se utilizará para conectarse a la base de datos MySQL
2 spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
3 # Define la URL de conexión a la base de datos (en este caso, una base local en el puerto 3307 con el nombre 'springbatch')
4 spring.datasource.url=jdbc:mysql://localhost:3307/springbatch
5 # Usuario de la base de datos que usará la aplicación para autenticarse
6 spring.datasource.username=root
7 # Contraseña correspondiente al usuario de la base de datos
8 spring.datasource.password=EzaicHV1109
9 # Muestra las sentencias SQL generadas por JPA/Hibernate en la consola durante la ejecución
10 spring.jpa.show-sql=true
11 # Configura la política de actualización del esquema de la base de datos
12 # (update: crea y actualiza tablas según las entidades sin borrar datos existentes)
13 spring.jpa.hibernate.ddl-auto=update
14 # Define el dialecto de Hibernate para MySQL 5, que le indica cómo generar el SQL compatible con esta versión
15 spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
16 # Establece el puerto en el que se ejecutará el servidor Spring Boot (por defecto es 8080, aquí se cambia a 9191)
17 server.port=9191
18 # Indica que el esquema de tablas de Spring Batch debe inicializarse automáticamente al arrancar la aplicación
19 spring.batch.initialize-schema=ALWAYS
20 # Evita que los jobs de Spring Batch se ejecuten automáticamente al iniciar la aplicación
21 spring.batch.job.enabled=false
```

## Estructura del Archivo CSV

El archivo CSV que se utiliza como entrada contiene datos estructurados de usuarios. Su formato es simple y separado por comas, con una cabecera en la primera línea.

Ejemplo:

	A	B	C	D	E	F	G	H
1	id	firstName	lastName	email	gender	contactNo	country	dob
2	1	Dud	Ishak	dishak0@	Male	970-613-2	Ukraine	#####
3	2	Jolyn	Bragg	jbragg1@g	Genderque	614-319-7	United Sta	#####

Cada línea representa un registro que será transformado en un objeto Java y almacenado en la base de datos si pasa las validaciones.

## Modelo de Datos

Los datos que se procesan desde el archivo CSV se transforman en objetos de la clase Customer, que a su vez está mapeada como entidad JPA. Esta clase representa un cliente y contiene atributos que reflejan la estructura de la tabla CUSTOMERS\_INFO en la base de datos. Se utilizó Lombok para generar automáticamente los métodos getters, setters, constructores y el método toString, lo que facilita el mantenimiento del código. A continuación, se muestra el modelo:

```

src > main > java > com > javatechie > spring > batch > entity > Customer.java > Customer > country
9  import javax.persistence.Id;
10 import javax.persistence.Table;
11
12 @Entity
13 @Table(name = "CUSTOMERS_INFO")
14 @Data
15 @AllArgsConstructor
16 @NoArgsConstructor
17 public class Customer {
18
19     @Id
20     @Column(name = "CUSTOMER_ID")
21     private int id;
22     @Column(name = "FIRST_NAME")
23     private String firstName;
24     @Column(name = "LAST_NAME")
25     private String lastName;
26     @Column(name = "EMAIL")
27     private String email;
28     @Column(name = "GENDER")
29     private String gender;
30     @Column(name = "CONTACT")
31     private String contactNo;
32     @Column(name = "COUNTRY")
33     private String country;
34     @Column(name = "DOB")
35     private String dob;
36

```

Cada uno de estos campos es mapeado directamente desde el archivo CSV mediante un `LineMapper`, y luego insertado en la base de datos MySQL usando el `ItemWriter`. La anotación `@Table` asegura que los datos se almacenen correctamente en la tabla `CUSTOMERS_INFO`, mientras que las anotaciones `@Column` especifican el nombre exacto de cada columna según la base de datos.

## Componentes de Spring Batch

El flujo del procesamiento está dividido en tres componentes principales, cada uno con una función específica:

### ItemReader

Se utiliza `FlatFileItemReader` para leer el archivo CSV línea por línea. Este lector está configurado para mapear automáticamente los campos del archivo a la entidad `Customer`, ignorando la cabecera del archivo. El mapeo se realiza a través de un `DelimitedLineTokenizer` y un `BeanWrapperFieldSetMapper`.

### ItemProcessor

Aquí se encuentra una lógica de filtrado específica. Se implementó una clase llamada `CustomerProcessor` que determina qué registros deben procesarse y cuáles no.

El procesador recibe un objeto `Customer` y lo analiza. En este caso, solo se permiten los registros cuyo campo `country` sea **"China"**. Si el valor es distinto, el objeto se descarta retornando `null`. Esto permite reducir los datos que se escriben en la base de datos y enfocarse solo en los registros relevantes. A continuación, se muestra la lógica implementada:

```

/**
 * Método que se ejecuta para cada objeto Customer leído del archivo.
 * Si el cliente es de "China", se retorna el objeto para ser escrito en la base de datos.
 * Si el cliente es de otro país, se retorna null, lo que significa que será ignorado.
 *
 * @param customer El objeto Customer leído desde el archivo CSV.
 * @return El mismo objeto si cumple la condición, o null si debe ser excluido.
 * @throws Exception En caso de error durante el procesamiento.
 */
@Override
public Customer process(Customer customer) throws Exception {
    // Verifica si el país del cliente es "China"
    if (customer.getCountry().equals(anObject:"China")) {
        // Si es de Estados Unidos, se acepta el objeto y se sigue el flujo
        return customer;
    } else {
        // Si es de otro país, se ignora (se retorna null y no se escribirá en la base de datos)
        return null;
    }
}
}

```

Este filtro es útil cuando se trabaja con archivos que contienen datos internacionales, pero solo se necesita conservar cierta información específica.

## ItemWriter

Se empleó un `JdbcBatchItemWriter` para insertar los datos validados en la base de datos. Este writer utiliza una consulta SQL predefinida y mapea los atributos del objeto `Customer` a las columnas correspondientes en la tabla `CUSTOMERS_INFO`.

El procesamiento se realiza en **chunks**, es decir, en bloques de registros (por ejemplo, 100 a la vez), lo que permite mejorar el rendimiento general y manejar volúmenes grandes de datos sin cargar toda la información en memoria.

## Resultados del Proceso

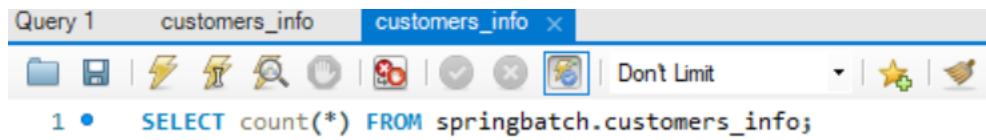
Para validar el funcionamiento del sistema, se procesó un archivo de entrada con un total de **5,000 registros**. El archivo contenía datos de clientes provenientes de distintos países.

Durante la ejecución del job, se obtuvieron los siguientes resultados:

- **Registros leídos desde el archivo CSV:** 1,000
- **Registros que cumplían la condición (country = "China"):** 191
- **Registros insertados en la base de datos:** 191
- **Registros ignorados por filtrado en el processor:** 0
- **Errores de lectura o escritura:** 0

Estos datos fueron verificados mediante la consola y los logs generados por Spring Batch, así como mediante consultas directas a la base de datos MySQL.

Este tipo de filtrado permite reducir significativamente la carga de datos en la base, manteniendo únicamente la información relevante según las reglas definidas en el ItemProcessor.



The screenshot shows a SQL query editor with a toolbar and a query text area. The toolbar includes icons for file operations, execution, and settings. The query text area contains the following SQL statement:

```
1 • SELECT count(*) FROM springbatch.customers_info;
```



The screenshot shows a database result grid with a toolbar. The toolbar includes icons for result grid, filter rows, export, and wrap cell content. The result grid displays the following data:

count(*)
191