Guia de Repaso

Spring Batch:
Procesamiento por Lotes

Semana 02 - Viernes 20 de Febrero de 2026

Academia Java - Ciudad de Mexico

Temas del dia:

1. Que es Spring Batch: procesamiento por lotes
2. Arquitectura: Job, Step, Reader-Processor-Writer
3. Chunk-oriented processing: procesar en bloques
4. Proyecto v1: CSV a MySQL (un Step)
5. Proyecto v2: Dos Steps (CSV a MySQL a CSV)
6. JobRepository: metadatos y control de ejecucion

Instructor: Miguel Rugerio

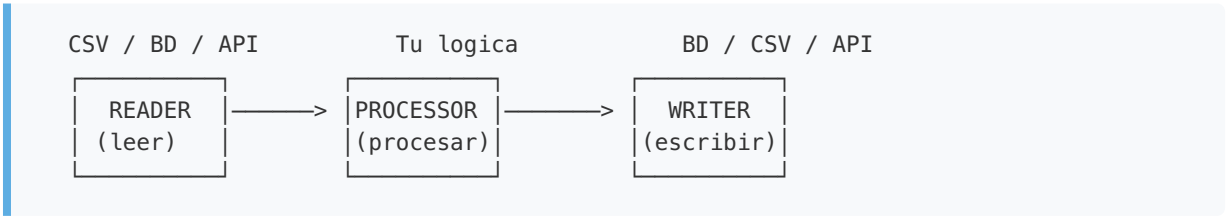
1. Que es Spring Batch

Spring Batch es un framework de Spring para **procesamiento por lotes** (batch processing). Procesa grandes cantidades de datos de forma automatizada, sin intervencion del usuario.

Analogia: la fabrica de empaquetado

Imagina una fabrica que recibe cajas de productos (datos de entrada), los inspecciona y etiqueta (procesamiento), y los coloca en un camion (datos de salida). La fabrica no procesa una caja a la vez, sino **grupos de cajas** (chunks) para ser eficiente.

Spring Batch funciona igual:



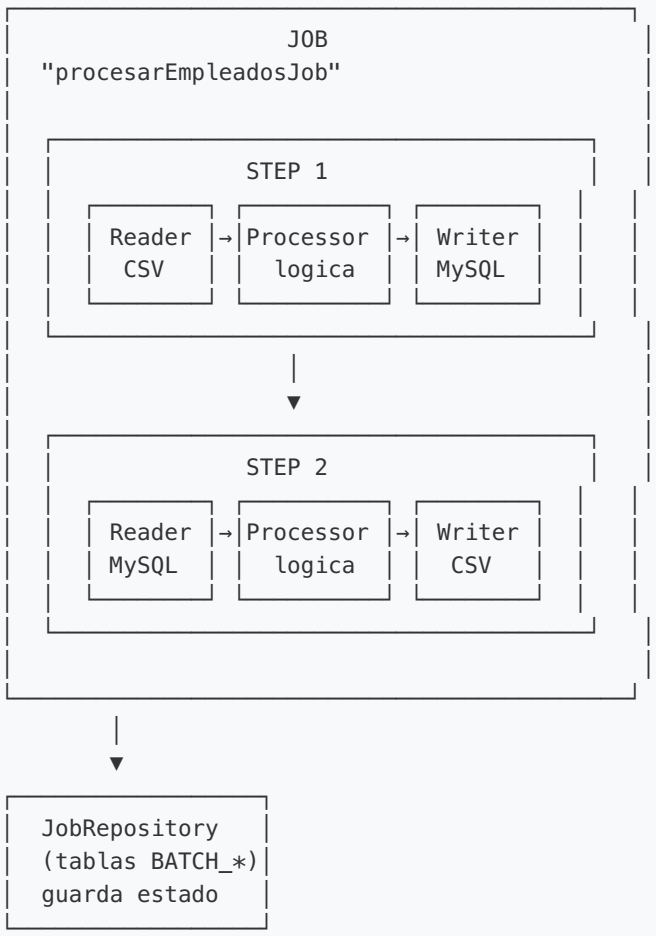
> **Clave:** Spring Batch se usa cuando necesitas procesar archivos grandes, migrar datos entre sistemas, generar reportes masivos, o cualquier tarea repetitiva sobre muchos registros.

Donde se usa en la industria

Escenario	Reader	Writer
Cargar empleados de un CSV a la base de datos	Archivo CSV	MySQL
Generar reporte de ventas del mes	Base de datos	Archivo CSV
Migrar datos entre sistemas	BD origen	BD destino
Procesar archivos de un banco	Archivo plano	Base de datos

2. Arquitectura de Spring Batch

Spring Batch tiene 3 niveles jerarquicos: **Job > Step > (Reader + Processor + Writer)**.



Los componentes

Componente	Que hace	Analogia
Job	El trabajo completo. Contiene uno o mas Steps	La orden de trabajo de la fabrica
Step	Un paso del Job. Contiene Reader + Processor + Writer	Una estacion de la linea de ensamblaje
ItemReader	Lee datos de una fuente (CSV, BD, API)	La banda que trae cajas
ItemProcessor	Transforma cada registro	El operario que inspecciona y etiqueta
ItemWriter	Escribe los resultados (BD, CSV, API)	El operario que pone cajas en el camion
JobRepository	Guarda el estado de cada ejecucion	El registro de produccion de la fabrica

> **Nota:** El Processor es **opcional**. Si solo necesitas copiar datos sin transformarlos, puedes tener solo Reader + Writer.

3. Chunk-oriented Processing

Spring Batch no procesa registro por registro. Procesa en **chunks** (bloques).

Como funciona con chunk(3)

Si tenemos 10 empleados y chunk size = 3:

```
Registros: [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]

Chunk 1: Lee 1, Lee 2, Lee 3  → Procesa 1,2,3  → Escribe [1,2,3] en BD
Chunk 2: Lee 4, Lee 5, Lee 6  → Procesa 4,5,6  → Escribe [4,5,6] en BD
Chunk 3: Lee 7, Lee 8, Lee 9  → Procesa 7,8,9  → Escribe [7,8,9] en BD
Chunk 4: Lee 10               → Procesa 10     → Escribe [10] en BD
```

> **Clave:** Cada chunk es una **transaccion**. Si el chunk 3 falla, los chunks 1 y 2 ya estan guardados en la base de datos. No se pierde todo el trabajo.

Por que no de 1 en 1?

Estrategia	INSERTs para 10 registros	Eficiencia
Uno por uno (chunk=1)	10 transacciones	Lento
Chunks de 3 (chunk=3)	**4 transacciones**	**Equilibrado**
Todo junto (chunk=10)	1 transaccion	Riesgo: si falla, se pierde todo

En nuestros proyectos usamos `chunk(3)` para poder ver claramente en los logs como se procesan los bloques.

4. Proyecto v1: CSV a MySQL (un Step)

El primer proyecto lee un archivo CSV de empleados, transforma los datos (nombre a mayusculas + calculo de bono), y los guarda en MySQL.

```
empleados.csv ──→ EmpleadoProcessor ──→ MySQL (empleados_procesados)
                  - MAYUSCULAS           INSERT INTO ...
                  - bono = salario*10%
```

El archivo CSV de entrada

```
nombre,departamento,salario
Juan Perez,Ventas,25000
Maria Lopez,TI,35000
Carlos Garcia,RRHH,28000
Ana Martinez,Ventas,27000
Pedro Sanchez,TI,32000
Laura Diaz,RRHH,30000
Roberto Flores,Ventas,26000
Sofia Ramirez,TI,38000
Diego Torres,RRHH,29000
Fernanda Rios,Ventas,31000
```

10 empleados con 3 campos: nombre, departamento, salario.

El modelo: Empleado (POJO)

```
public class Empleado {

    private String nombre;
    private String departamento;
    private double salario;
    private double bono;

    public Empleado() {
    }

    // getters y setters para cada campo...

    @Override
    public String toString() {
        return nombre + " | " + departamento
             + " | Salario: " + salario + " | Bono: " + bono;
    }
}
```

Clave: Spring Batch usa los **setters** para llenar el objeto al leer el CSV, y los **getters** para obtener los valores al escribir en la BD. Por eso necesita el constructor vacio y los getters/setters.

5. El Reader: leer el CSV

El `FlatFileItemReader` lee archivos planos (CSV, TXT) linea por linea:

```
@Bean
public FlatFileItemReader<Empleado> leerCsv() {
    return new FlatFileItemReaderBuilder<Empleado>()
            .name("empleadoReader")
            .resource(new ClassPathResource("empleados.csv"))
            .delimited()                          // separado por comas
            .names("nombre", "departamento", "salario") // columnas del CSV
            .targetType(Empleado.class)            // mapea a nuestro POJO
            .linesToSkip(1)                        // saltar encabezado
            .build();
}
```

Que hace cada linea

Metodo	Que hace
`.name("empleadoReader")`	Nombre interno para el reader
`.resource(new ClassPathResource(...))`	Busca el archivo en `src/main/resources/`
`.delimited()`	Indica que el separador es coma (,)
`.names(...)`	Mapea columnas del CSV a los setters del POJO
`.targetType(Empleado.class)`	Tipo de objeto a crear por cada linea
`.linesToSkip(1)`	Salta la primera linea (el encabezado)

Nota: `ClassPathResource` busca archivos dentro de `src/main/resources/`. Es donde Maven y Spring Boot esperan encontrar archivos estaticos del proyecto.

6. El Processor: transformar los datos

El `ItemProcessor` recibe un objeto, lo transforma, y lo devuelve:

```
public class EmpleadoProcessor implements ItemProcessor<Empleado, Empleado> {

    private static final Logger log =
LoggerFactory.getLogger(EmpleadoProcessor.class);

    @Override
    public Empleado process(Empleado empleado) {
        empleado.setNombre(empleado.getNombre().toUpperCase());
        empleado.setBono(empleado.getSalario() * 0.10);

        log.info("Step 1 - Procesando: {}", empleado);
        return empleado;
    }
}
```

Que hace

1. **Nombre a mayusculas:** `"Juan Perez"` → `"JUAN PEREZ"`
2. **Calcula bono del 10%:** salario 25000 → bono 2500.0
3. **Log:** imprime en consola cada empleado procesado

La interfaz ItemProcessor<I, O>

```
ItemProcessor<Empleado, Empleado>
                  │          │
                  │          └── O = tipo de SALIDA (lo que devuelve)
                  └── I = tipo de ENTRADA (lo que recibe del Reader)
```

En v1, la entrada y salida son el mismo tipo (`Empleado`). En v2 veremos que pueden ser diferentes.

> **Clave:** Si `process()` retorna `null` , el registro se **descarta** y no llega al Writer. Esto es util para filtrar datos.

7. El Writer: escribir en MySQL

El `JdbcBatchItemWriter` inserta los registros procesados en la base de datos:

```
@Bean
public JdbcBatchItemWriter<Empleado> escribirEnBd(DataSource dataSource) {
    return new JdbcBatchItemWriterBuilder<Empleado>()
            .sql("INSERT INTO empleados_procesados "
               + "(nombre, departamento, salario, bono) "
               + "VALUES (:nombre, :departamento, :salario, :bono)")
            .dataSource(dataSource)
            .beanMapped()
            .build();
}
```

Los parametros con nombre (:nombre, :salario)

```
VALUES (:nombre, :departamento, :salario, :bono)
        |          |             |         |
        ▼          ▼             ▼         ▼
  getNombre() getDepartamento() getSalario() getBono()
```

> **Clave:** `.beanMapped()` le dice a Spring Batch: "los parametros `:nombre`, `:salario`, etc. se obtienen llamando `getNombre()`, `getSalario()` del POJO". Por eso los nombres deben coincidir exactamente.

DataSource: la conexion a MySQL

`DataSource` es el objeto que contiene la conexion a la base de datos. **No lo creamos nosotros** — Spring Boot lo crea automaticamente usando el `application.properties`:

```
spring.datasource.url=jdbc:mysql://localhost:3306/academia
spring.datasource.username=alumno
spring.datasource.password=alumno123
```

Spring Boot ve estas propiedades, crea un `DataSource`, y lo inyecta donde se necesite. Esto es **Inversion de Control** en accion.

8. El Step y el Job: ensamblando las piezas

El Step conecta Reader + Processor + Writer

```
@Bean
public Step paso1(JobRepository jobRepository,
                  PlatformTransactionManager transactionManager,
                  FlatFileItemReader<Empleado> leerCsv,
                  EmpleadoProcessor procesarEmpleado,
                  JdbcBatchItemWriter<Empleado> escribirEnBd) {

    return new StepBuilder("paso1", jobRepository)
            .<Empleado, Empleado>chunk(3, transactionManager)
            .reader(leerCsv)
            .processor(procesarEmpleado)
            .writer(escribirEnBd)
            .build();
}
```

Que es <Empleado, Empleado>chunk(3, transactionManager)?

```
.<Empleado, Empleado>chunk(3, transactionManager)
   |          |        |    |
   |          |        |    └── maneja las transacciones (commit/rollback)
   |          |        └── tamanio del chunk (procesa de 3 en 3)
   |          └── tipo de SALIDA del processor
   └── tipo de ENTRADA al processor
```

El Job ejecuta los Steps

```
@Bean
public Job procesarEmpleadosJob(JobRepository jobRepository, Step paso1) {
    return new JobBuilder("procesarEmpleadosJob", jobRepository)
            .start(paso1)   // inicia con el paso1
            .build();
}
```

> **Nota:** `JobRepository`, `PlatformTransactionManager` y `DataSource` son **inyectados automaticamente** por Spring Boot. No necesitas crearlos manualmente — Spring Boot los configura usando las propiedades de `application.properties`.

9. @Configuration y @Bean: la receta de la fabrica

@Configuration

Marca una clase como **fuente de configuracion**. Le dice a Spring: "esta clase contiene la receta para crear objetos":

```
@Configuration
public class BatchConfig {
    // aqui van los @Bean
}
```

@Bean

Cada metodo con `@Bean` le dice a Spring: "ejecuta este metodo y guarda el resultado para cuando alguien lo necesite":

```
@Bean
public FlatFileItemReader<Empleado> leerCsv() {
    // Spring ejecuta esto UNA vez y guarda el reader
    return new FlatFileItemReaderBuilder<Empleado>()
            // ...
            .build();
}

@Bean
public Step paso1(..., FlatFileItemReader<Empleado> leerCsv, ...) {
    // Spring inyecta el reader que creo arriba
    return new StepBuilder("paso1", jobRepository)
            .reader(leerCsv)  // <-- usa el reader inyectado
            // ...
            .build();
}
```

Clave: Los parametros de un metodo `@Bean` se resuelven por **inyeccion de dependencias**. Spring busca un bean del tipo correcto y lo pasa automaticamente. Por eso `paso1` recibe el `leerCsv` sin que nosotros llamemos `new`.

Relacion con IoC (sesion anterior)

Ayer vimos IoC manual con `setPizza()` y luego un `Inyector` centralizado. **Spring Boot es el inyector real**: busca los `@Bean` que definimos y los conecta donde se necesiten, usando `@Configuration` como el mapa de instrucciones.

10. JobRepository: el registro de produccion

Spring Batch crea automaticamente **9 tablas** en MySQL con prefijo `BATCH_`:

```
BATCH_JOB_INSTANCE        ← un Job unico (nombre + parametros)
BATCH_JOB_EXECUTION       ← cada ejecucion del Job (fecha, estado, duracion)
BATCH_JOB_EXECUTION_PARAMS ← parametros de cada ejecucion
BATCH_STEP_EXECUTION       ← cada ejecucion de cada Step (registros leidos,
escritos)
...y 5 tablas mas de soporte
```

Para que sirven?

```
-- Ver las ejecuciones del Job
SELECT * FROM BATCH_JOB_EXECUTION;

-- Ver cuantos registros proceso cada Step
SELECT step_name, read_count, write_count, status
FROM BATCH_STEP_EXECUTION;
```

Campo	Ejemplo	Significado
`status`	COMPLETED	El Step termino correctamente
`read_count`	10	Leyo 10 registros del CSV
`write_count`	10	Escribio 10 registros en MySQL
`commit_count`	4	Hizo 4 commits (10 registros / chunk 3 = 4 chunks)

Problema: Si ejecutas el Job dos veces con los mismos parametros, Spring Batch dice: "ya lo ejecute, no lo vuelvo a hacer". Si necesitas re-ejecutar, primero limpia las tablas BATCH_* y la tabla de datos.

11. Proyecto v2: Dos Steps

La version 2 agrega un segundo Step que lee los datos ya procesados de MySQL y genera un reporte CSV con el salario total:

```
  STEP 1                          STEP 2
  CSV ──→ Processor ──→ MySQL     MySQL ──→ Processor ──→ CSV
          - MAYUSCULAS                      - salarioTotal
          - bono 10%                        = salario + bono
```

El Job ejecuta los Steps en secuencia

```
@Bean
public Job procesarEmpleadosJob(JobRepository jobRepository,
                                Step paso1, Step paso2) {
    return new JobBuilder("procesarEmpleadosJob", jobRepository)
            .start(paso1)       // primero ejecuta paso1
            .next(paso2)        // despues ejecuta paso2
            .build();
}
```

Clave: `.start(paso1).next(paso2)` define el orden. Si paso1 falla, paso2 **no se ejecuta**.

12. Step 2: nuevo modelo y processor

EmpleadoReporte: un modelo diferente

El Step 2 necesita un campo extra: `salarioTotal`. Creamos un nuevo POJO:

```
public class EmpleadoReporte {

    private String nombre;
    private String departamento;
    private double salario;
    private double bono;
    private double salarioTotal;    // <-- campo nuevo

    // constructor vacio + getters y setters para todos los campos
}
```

ReporteProcessor: tipos diferentes de entrada y salida

```
public class ReporteProcessor
        implements ItemProcessor<Empleado, EmpleadoReporte> {

    private static final Logger log =
        LoggerFactory.getLogger(ReporteProcessor.class);

    @Override
    public EmpleadoReporte process(Empleado empleado) {
        EmpleadoReporte reporte = new EmpleadoReporte();
        reporte.setNombre(empleado.getNombre());
        reporte.setDepartamento(empleado.getDepartamento());
        reporte.setSalario(empleado.getSalario());
        reporte.setBono(empleado.getBono());
        reporte.setSalarioTotal(
            empleado.getSalario() + empleado.getBono());

        log.info("Step 2 - Reporte: {}", reporte);
        return reporte;
    }
}
```

Tipos diferentes: Empleado entra, EmpleadoReporte sale

```
ItemProcessor<Empleado, EmpleadoReporte>
                  |              |
                  |              └── SALE un EmpleadoReporte (con salarioTotal)
                  └── ENTRA un Empleado (leido de MySQL)
```

Clave: En v1 el processor era `<Empleado, Empleado>` (mismo tipo). En v2 es `<Empleado, EmpleadoReporte>` (tipos diferentes). Esto demuestra que el processor puede **transformar** el tipo de dato, no solo modificarlo.

13. Step 2: Reader de base de datos

JdbcCursorItemReader: leer de MySQL

```
@Bean
public JdbcCursorItemReader<Empleado> leerDeBd(DataSource dataSource) {
    return new JdbcCursorItemReaderBuilder<Empleado>()
            .name("empleadoDbReader")
            .dataSource(dataSource)
            .sql("SELECT nombre, departamento, salario, bono "
               + "FROM empleados_procesados")
            .rowMapper((rs, rowNum) -> {
                Empleado empleado = new Empleado();
                empleado.setNombre(rs.getString("nombre"));
                empleado.setDepartamento(rs.getString("departamento"));
                empleado.setSalario(rs.getDouble("salario"));
                empleado.setBono(rs.getDouble("bono"));
                return empleado;
            })
            .build();
}
```

El rowMapper: convertir filas SQL a objetos Java

El `rowMapper` es una funcion que Spring llama **por cada fila** del resultado SQL:

```
Fila SQL: | JUAN PEREZ | VENTAS | 25000.0 | 2500.0 |
                │            │         │         │
                ▼            ▼         ▼         ▼
      rs.getString   rs.getString  rs.getDouble  rs.getDouble
       ("nombre")  ("departamento") ("salario")    ("bono")
                │            │         │         │
                ▼            ▼         ▼         ▼
         setNombre()  setDepartamento() setSalario() setBono()
                │            │         │         │
                └────────────┴─────────┴─────────┘
                             │
                      Empleado completo
```

> **Nota:** Usamos una **lambda** porque `RowMapper` es una interfaz funcional (tiene un solo metodo abstracto). Es lo mismo que vimos ayer con `Runnable` y los threads.

14. Step 2: Writer a archivo CSV

FlatFileItemWriter: escribir un CSV de salida

```
@Bean
public FlatFileItemWriter<EmpleadoReporte> escribirCsv() {
    return new FlatFileItemWriterBuilder<EmpleadoReporte>()
            .name("reporteWriter")
            .resource(new FileSystemResource("reporte-empleados.csv"))
            .headerCallback(writer -> writer.write(
                "nombre,departamento,salario,bono,salario_total"))
            .delimited()
            .names("nombre", "departamento", "salario",
                   "bono", "salarioTotal")
            .build();
}
```

ClassPathResource vs FileSystemResource

Tipo	Donde busca	Uso tipico
`ClassPathResource`	Dentro de `src/main/resources/`	Archivos de **entrada** empaquetados en el proyecto
`FileSystemResource`	En el **disco duro** (raiz del proyecto)	Archivos de **salida** generados por la aplicacion

Clave: El CSV de entrada (`empleados.csv`) se lee con `ClassPathResource` porque es parte del proyecto. El CSV de salida (`reporte-empleados.csv`) se escribe con `FileSystemResource` porque es un archivo **generado** en tiempo de ejecucion.

El archivo generado

```
nombre,departamento,salario,bono,salario_total
JUAN PEREZ,VENTAS,25000.0,2500.0,27500.0
MARIA LOPEZ,TI,35000.0,3500.0,38500.0
CARLOS GARCIA,RRHH,28000.0,2800.0,30800.0
...
```

> **Nota en Eclipse:** El archivo `reporte-empleados.csv` aparece en la raiz del proyecto. Si no lo ves, haz clic derecho en el proyecto → **Refresh** (o presiona **F5**).

15. La configuracion completa de v2

BatchConfig.java con los dos Steps

```
@Configuration
public class BatchConfig {

    // =============== STEP 1: CSV → MySQL ===============

    @Bean
    public FlatFileItemReader<Empleado> leerCsv() { ... }

    @Bean
    public EmpleadoProcessor procesarEmpleado() { ... }

    @Bean
    public JdbcBatchItemWriter<Empleado> escribirEnBd(DataSource ds) { ... }

    @Bean
    public Step paso1(...) {
        return new StepBuilder("paso1", jobRepository)
                .<Empleado, Empleado>chunk(3, tx)   // mismo tipo
                .reader(leerCsv)
                .processor(procesarEmpleado)
                .writer(escribirEnBd)
                .build();
    }

    // =============== STEP 2: MySQL → CSV ===============

    @Bean
    public JdbcCursorItemReader<Empleado> leerDeBd(DataSource ds) { ... }

    @Bean
    public ReporteProcessor procesarReporte() { ... }

    @Bean
    public FlatFileItemWriter<EmpleadoReporte> escribirCsv() { ... }

    @Bean
    public Step paso2(...) {
        return new StepBuilder("paso2", jobRepository)
                .<Empleado, EmpleadoReporte>chunk(3, tx)  // tipos diferentes!
                .reader(leerDeBd)
                .processor(procesarReporte)
                .writer(escribirCsv)
                .build();
    }

    // =============== JOB ===============

    @Bean
    public Job procesarEmpleadosJob(JobRepository repo,
                                    Step paso1, Step paso2) {
```

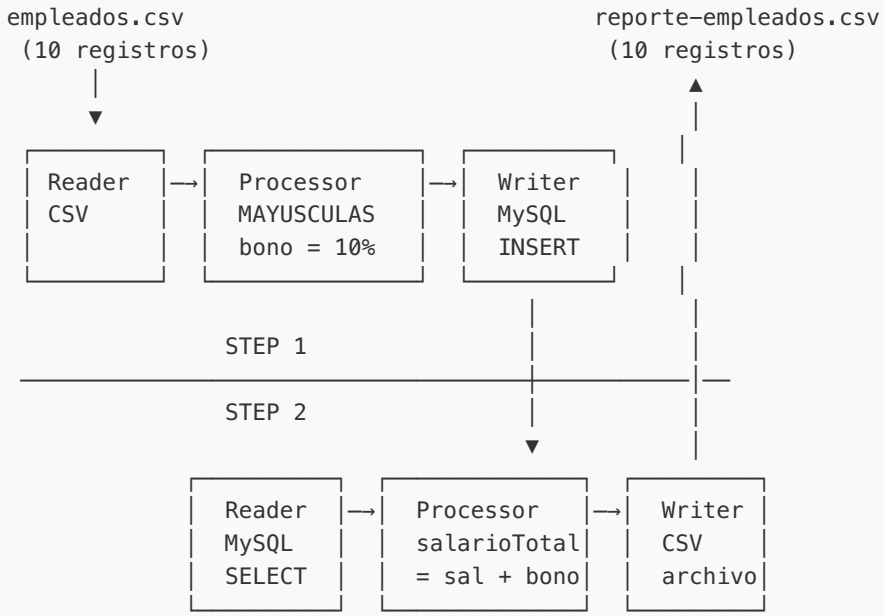
```
        return new JobBuilder("procesarEmpleadosJob", repo)
                .start(paso1)
                .next(paso2)
                .build();
    }
}
```

16. Comparacion: v1 vs v2

Componentes por proyecto

Componente	v1	v2
Steps	1	2
Readers	FlatFileItemReader (CSV)	+ JdbcCursorItemReader (BD)
Processors	EmpleadoProcessor	+ ReporteProcessor
Writers	JdbcBatchItemWriter (BD)	+ FlatFileItemWriter (CSV)
Modelos	Empleado	+ EmpleadoReporte
Tipos del Processor	`<Empleado, Empleado>`	+ `<Empleado, EmpleadoReporte>`

Flujo completo de v2



17. application.properties explicado

```
# Conexion a MySQL (contenedor docker mysql-academia)
spring.datasource.url=jdbc:mysql://localhost:3306/academia
spring.datasource.username=alumno
spring.datasource.password=alumno123
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# Spring Batch crea automaticamente las tablas BATCH_* en MySQL
spring.batch.jdbc.initialize-schema=always

# Ejecutar el Job automaticamente al iniciar la aplicacion
spring.batch.job.enabled=true
```

Que crea Spring Boot con estas propiedades

Propiedad	Que hace Spring Boot
`spring.datasource.*`	Crea un `DataSource` (conexion a MySQL) y lo inyecta donde se necesite
`spring.batch.jdbc.initialize-schema=always`	Crea las 9 tablas `BATCH_*` automaticamente al iniciar
`spring.batch.job.enabled=true`	Ejecuta el Job inmediatamente al arrancar la aplicacion

Clave: Con solo 6 lineas de configuracion, Spring Boot crea automaticamente: el `DataSource`, el `JobRepository`, el `PlatformTransactionManager`, y las tablas de metadatos. Sin estas propiedades, tendriamos que crear todo manualmente.

18. Dependencias Maven

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-batch</artifactId>
    </dependency>
    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

Solo 2 dependencias:

Dependencia	Que incluye
`spring-boot-starter-batch`	Spring Batch + Spring Boot + Spring JDBC + transacciones
`mysql-connector-j`	Driver JDBC para conectarse a MySQL

Nota: `spring-boot-starter-batch` incluye todo lo que necesitamos. Spring Boot se encarga de versiones compatibles gracias al `<parent>` en el `pom.xml`. No necesitamos especificar versiones individuales.

La tabla SQL necesaria

Antes de ejecutar, debes crear la tabla destino en MySQL:

```
CREATE TABLE IF NOT EXISTS empleados_procesados (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(100),
    departamento VARCHAR(50),
    salario DOUBLE,
    bono DOUBLE
);
```

Las tablas `BATCH_*` las crea Spring Batch automaticamente gracias a la propiedad `spring.batch.jdbc.initialize-schema=always`.

Resumen de conceptos clave

Concepto	Descripcion
Spring Batch	Framework para procesamiento por lotes (grandes volumenes de datos)
Job	El trabajo completo. Contiene uno o mas Steps
Step	Un paso: Reader + Processor + Writer
`ItemReader`	Lee datos de una fuente (CSV, BD)
`ItemProcessor`	Transforma cada registro. Puede cambiar el tipo de dato
`ItemWriter`	Escribe los resultados (BD, CSV)
`chunk(N)`	Procesa N registros por transaccion
`JobRepository`	Tablas `BATCH_*` que guardan el estado de cada ejecucion
`@Configuration`	Marca la clase como fuente de beans (recetas)
`@Bean`	Marca un metodo cuyo resultado Spring guarda e inyecta
`ClassPathResource`	Busca archivos dentro de `src/main/resources/`
`FileSystemResource`	Busca/crea archivos en el disco (fuera del jar)
`beanMapped()`	Mapea `:parametros` SQL a los getters del POJO
`rowMapper`	Funcion que convierte cada fila SQL en un objeto Java
`.start().next()`	Define el orden de ejecucion de los Steps

Progresion de los ejercicios del dia

Proyecto	Paquete	Version	Tema
springBatch	`com.academia.batch`	v1	CSV → MySQL (1 Step)
springBatch	`com.academia.batch`	v1	`FlatFileItemReader` lee CSV
springBatch	`com.academia.batch`	v1	`EmpleadoProcessor` transforma datos
springBatch	`com.academia.batch`	v1	`JdbcBatchItemWriter` escribe en BD
springBatch	`com.academia.batch`	v1	`chunk(3)` procesamiento por bloques
springBatchV2	`com.academia.batch`	v2	Dos Steps: `.start().next()`
springBatchV2	`com.academia.batch`	v2	`JdbcCursorItemReader` lee de BD
springBatchV2	`com.academia.batch`	v2	`ReporteProcessor` tipos diferentes
springBatchV2	`com.academia.batch`	v2	`FlatFileItemWriter` genera CSV
springBatchV2	`com.academia.batch`	v2	`ClassPathResource` vs `FileSystemResource`