Guía de Construcción - Spring Batch para Principiantes

Procesar Millones de Registros Más Rápido con Spring Batch

¿Qué es Spring Batch?

Spring Batch es uno de los módulos principales del framework Spring que te permite crear sistemas robustos de procesamiento por lotes. Con Spring Batch puedes procesar grandes volúmenes de datos en una fracción de tiempo.

¿Qué es Batch Processing?

Batch Processing es una técnica que procesa datos en grandes grupos en lugar de elemento por elemento, permitiendo procesar altos volúmenes de datos con mínima intervención humana.

Casos de Uso Principales

1. **Análisis de Edificios**: Tienes información de edificios en formato CSV y quieres volcarlo a una base de datos
   - **Fuente**: Archivo CSV
   - **Destino**: Base de datos
2. **Generación de Reportes**: Exportar reportes diarios CSV/Excel desde la base de datos
   - **Fuente**: Base de datos
   - **Destino**: Archivo CSV/Excel

Descripción del Proyecto Actual

Este proyecto demuestra un sistema completo de ETL (Extract, Transform, Load) que:

1. **Step 1**: Lee un archivo CSV con 1000+ registros de clientes, filtra solo mujeres de China, y los guarda en MySQL
2. **Step 2**: Lee los datos de MySQL, transforma los emails agregando dominio "@china.com", y los migra a MongoDB

Arquitectura de Spring Batch

Componentes Principales y Flujo de Ejecución

```
JobLauncher → Job → JobRepository
                ↓
              Step → ItemReader → ItemProcessor → ItemWriter
```

1. **Job Launcher**
   - **Propósito**: Punto de entrada para iniciar cualquier job en Spring Batch
   - **Función**: Interface que lanza jobs de Spring Batch
   - **Método Principal**: `run()` que dispara el objeto Job

2. **Job**
   - **Propósito**: Define el trabajo a ejecutar usando Spring Batch
   - **Función**: Puede involucrar tareas simples o complejas
   - **Relación**: Un Job puede tener múltiples Steps

3. **Job Repository**
   - **Propósito**: Mantiene el estado del job (éxito o fallo)
   - **Función**: Gestión de estado importante al procesar grandes volúmenes de datos
   - **Importancia**: Si ocurre un error, Spring Batch sabe que el job necesita reiniciarse

4. **Step**
   - **Propósito**: Combinación de ItemReader, ItemProcessor e ItemWriter
   - **Función**: Unidad de procesamiento dentro de un Job
   - **Flexibilidad**: Un Job puede tener múltiples Steps

5. **Item Reader**
   - **Propósito**: Lee datos de la fuente
   - **En nuestro proyecto**: Lee el archivo CSV `customers.csv`

6. **Item Processor**
   - **Propósito**: Procesa los datos entre lectura y escritura
   - **En nuestro proyecto**:
     - Step 1: Filtra solo mujeres de China
     - Step 2: Transforma emails agregando "@china.com"

7. **Item Writer**
   - **Propósito**: Escribe datos al destino
   - **En nuestro proyecto**:
     - Step 1: Guarda en MySQL
     - Step 2: Guarda en MongoDB

Flujo de Nuestro Proyecto

```
CSV File → FlatFileItemReader → CustomerProcessor → RepositoryItemWriter → MySQL
                                                                              ↓
MongoDB ← RepositoryItemWriter ← EmailDomainProcessor ← RepositoryItemReader ← MySQL
```

Prerrequisitos del Sistema

Software Necesario

- **Java 8 o superior**
- **Maven 3.6+**
- **MySQL 5.7+**
- **MongoDB 4.0+**
- **IDE** (IntelliJ IDEA, Eclipse, VS Code, etc.)

Bases de Datos

MySQL

- **Host**: localhost:3306
- **Base de datos**: `javatechie`
- **Usuario**: `root`
- **Contraseña**: `xideral1234`

MongoDB

- **URI**: `mongodb://admin:xideral4321@localhost:27017/empleado?authSource=admin`
- **Colección**: `customers`

Estructura del Proyecto

```
springBatchV2/
  src/
      main/
          java/com/javatechie/spring/batch/
              config/
                  SpringBatchConfig.java       # Configuración principal de Spring Batch
                  CustomerProcessor.java       # Procesador para filtrar clientes
                  EmailDomainProcessor.java    # Procesador para transformar emails
              controller/
                  JobController.java           # Controlador REST para ejecutar jobs
              entity/
                  Customer.java                # Entidad JPA para MySQL
                  CustomerMongo.java           # Entidad para MongoDB
              repository/
                  CustomerRepository.java      # Repositorio MySQL
                  CustomerMongoRepository.java # Repositorio MongoDB
```

```
            BatchProcessingDemoApplication.java # Clase principal
        resources/
            application.properties          # Configuración de la aplicación
            customers.csv                   # Archivo de datos CSV
    test/
pom.xml                                      # Configuración Maven
README.md
```

Pasos para Construir el Proyecto (Tutorial Paso a Paso)

1. Crear Proyecto Spring Boot

Dependencias Necesarias en pom.xml

```xml
<dependencies>
    <!-- Spring Boot Starter Batch -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-batch</artifactId>
    </dependency>

    <!-- Spring Boot Starter Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- Spring Boot Starter Web -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- Spring Boot Starter Data MongoDB -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-mongodb</artifactId>
    </dependency>

    <!-- MySQL Connector -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>

    <!-- Lombok -->
```

```
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>
```

2. Preparación del Entorno

Configurar MySQL

```
-- Crear la base de datos
CREATE DATABASE javatechie;

-- Crear usuario (si es necesario)
CREATE USER 'root'@'localhost' IDENTIFIED BY 'xideral1234';
GRANT ALL PRIVILEGES ON javatechie.* TO 'root'@'localhost';
FLUSH PRIVILEGES;
```

Configurar MongoDB

```
# Iniciar MongoDB con autenticación
mongod --auth --port 27017

# Crear usuario administrador
mongo admin
db.createUser({
  user: "admin",
  pwd: "xideral4321",
  roles: ["userAdminAnyDatabase", "dbAdminAnyDatabase", "readWriteAnyDatabase"]
})

# Crear base de datos empleado
use empleado
```

3. Crear Entidades

Customer.java (para MySQL)

```
@Entity
@Table(name = "CUSTOMERS_INFO")
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Customer {
    @Id
    @Column(name = "CUSTOMER_ID")
    private int id;
```

```
    @Column(name = "FIRST_NAME")
    private String firstName;

    @Column(name = "LAST_NAME")
    private String lastName;

    @Column(name = "EMAIL")
    private String email;

    @Column(name = "GENDER")
    private String gender;

    @Column(name = "CONTACT")
    private String contactNo;

    @Column(name = "COUNTRY")
    private String country;

    @Column(name = "DOB")
    private String dob;
}
```

CustomerMongo.java (para MongoDB)

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Document(collection = "customers")
public class CustomerMongo {
    @Id
    private String id;
    private Integer customerId;
    private String firstName;
    private String lastName;
    private String email;
    private String gender;
    private String contactNo;
    private String country;
    private String dob;
}
```

4. Crear Repositorios

CustomerRepository.java

```java
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Integer> {
}
```

CustomerMongoRepository.java

```java
@Repository
public interface CustomerMongoRepository extends MongoRepository<CustomerMongo, String> {
}
```

5. Configurar application.properties

```
# Configuración MySQL
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/javatechie
spring.datasource.username=root
spring.datasource.password=xideral1234
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

# Configuración servidor
server.port=9191

# Configuración Spring Batch
spring.batch.initialize-schema=ALWAYS
spring.batch.job.enabled=false  # No ejecutar jobs automáticamente

# Configuración MongoDB
spring.data.mongodb.uri=mongodb://admin:xideral4321@localhost:27017/empleado?authSource=admi
```

6. Crear Configuración de Spring Batch

SpringBatchConfig.java Paso 1: Anotaciones y Dependencias

```java
@Configuration
@EnableBatchProcessing
@AllArgsConstructor
public class SpringBatchConfig {

    private JobBuilderFactory jobBuilderFactory;
    private StepBuilderFactory stepBuilderFactory;
    private CustomerRepository customerRepository;
    private CustomerMongoRepository customerMongoRepository;
```

Paso 2: Crear Item Reader (Lee CSV)

```java
@Bean
public FlatFileItemReader<Customer> reader() {
    FlatFileItemReader<Customer> itemReader = new FlatFileItemReader<>();
    itemReader.setResource(new FileSystemResource("src/main/resources/customers.csv"));
    itemReader.setName("csvReader");
    itemReader.setLinesToSkip(1);  // Saltar header
    itemReader.setLineMapper(lineMapper());
    return itemReader;
}

private LineMapper<Customer> lineMapper() {
    DefaultLineMapper<Customer> lineMapper = new DefaultLineMapper<>();

    DelimitedLineTokenizer lineTokenizer = new DelimitedLineTokenizer();
    lineTokenizer.setDelimiter(",");
    lineTokenizer.setStrict(false);
    lineTokenizer.setNames("id", "firstName", "lastName", "email", "gender", "contactNo", "
    BeanWrapperFieldSetMapper<Customer> fieldSetMapper = new BeanWrapperFieldSetMapper<>();
    fieldSetMapper.setTargetType(Customer.class);

    lineMapper.setLineTokenizer(lineTokenizer);
    lineMapper.setFieldSetMapper(fieldSetMapper);

    return lineMapper;
}
```

Paso 3: Crear Item Writers

```java
@Bean
public RepositoryItemWriter<Customer> writer() {
    RepositoryItemWriter<Customer> writer = new RepositoryItemWriter<>();
    writer.setRepository(customerRepository);
    writer.setMethodName("save");
    return writer;
}

@Bean
public RepositoryItemWriter<CustomerMongo> mongoWriter() {
    RepositoryItemWriter<CustomerMongo> writer = new RepositoryItemWriter<>();
    writer.setRepository(customerMongoRepository);
    writer.setMethodName("save");
    return writer;
}
```

Paso 4: Crear Steps

```java
@Bean
```

```
public Step step1() {
    return stepBuilderFactory.get("csv-step").<Customer, Customer>chunk(10)
            .reader(reader())
            .processor(processor())
            .writer(writer())
            .taskExecutor(taskExecutor())  // Para procesamiento paralelo
            .build();
}

@Bean
public Step step2() {
    return stepBuilderFactory.get("second-step").<Customer, CustomerMongo>chunk(10)
            .reader(databaseReader())
            .processor(emailDomainProcessor())
            .writer(mongoWriter())
            .build();
}
```

Paso 5: Crear Job

```
@Bean
public Job runJob() {
    return jobBuilderFactory.get("importCustomers")
            .flow(step1())
            .next(step2())
            .end().build();
}
```

7. Crear Procesadores

CustomerProcessor.java (Filtra solo mujeres de China)

```
public class CustomerProcessor implements ItemProcessor<Customer, Customer> {
    @Override
    public Customer process(Customer customer) throws Exception {
        if(customer.getCountry().equals("China") &&
           customer.getGender().equals("Female")) {
            return customer;  // Procesar solo si cumple condición
        } else {
            return null;      // Filtrar fuera
        }
    }
}
```

EmailDomainProcessor.java (Transforma emails)

```
public class EmailDomainProcessor implements ItemProcessor<Customer, CustomerMongo> {
    @Override
```

```
    public CustomerMongo process(Customer customer) throws Exception {
        CustomerMongo customerMongo = new CustomerMongo();

        // Mapear campos
        customerMongo.setCustomerId(customer.getId());
        customerMongo.setFirstName(customer.getFirstName());
        customerMongo.setLastName(customer.getLastName());
        customerMongo.setGender(customer.getGender());
        customerMongo.setContactNo(customer.getContactNo());
        customerMongo.setCountry(customer.getCountry());
        customerMongo.setDob(customer.getDob());

        // Transformar email
        if (customer.getEmail() != null && customer.getEmail().contains("@")) {
            String emailPart = customer.getEmail().substring(0, customer.getEmail().indexOf
            String newEmail = emailPart + "@china.com";
            customerMongo.setEmail(newEmail);
        }

        return customerMongo;
    }
}
```

8. Crear Controlador para Ejecutar Jobs

JobController.java

```
@RestController
@RequestMapping("/jobs")
public class JobController {

    @Autowired
    private JobLauncher jobLauncher;

    @Autowired
    private Job job;

    @PostMapping("/importCustomers")
    public void importCsvToDBJob() {
        JobParameters jobParameters = new JobParametersBuilder()
                .addLong("startAt", System.currentTimeMillis())
                .toJobParameters();

        try {
            jobLauncher.run(job, jobParameters);
        } catch (JobExecutionAlreadyRunningException | JobRestartException |
```

```
                JobInstanceAlreadyCompleteException | JobParametersInvalidException e) {
            e.printStackTrace();
        }
    }
}
```

9. Optimización de Rendimiento con Task Executor

¿Por qué optimizar? Por defecto, Spring Batch es **síncrono**. Si procesas 1000 registros uno por uno, puede tomar mucho tiempo. La solución es usar **Task Executor** para procesamiento **asíncrono** y **paralelo**.

Agregar Task Executor a SpringBatchConfig.java

```
@Bean
public TaskExecutor taskExecutor() {
    SimpleAsyncTaskExecutor asyncTaskExecutor = new SimpleAsyncTaskExecutor();
    asyncTaskExecutor.setConcurrencyLimit(10);  // 10 hilos paralelos
    return asyncTaskExecutor;
}
```

Resultado de Rendimiento: - **Sin Task Executor**: ~6 segundos para 1000 registros - **Con Task Executor (10 hilos)**: ~2-3 segundos para 1000 registros

10. Instalación y Compilación

```
# Navegar al directorio del proyecto
cd /Users/mike/Desarrollo/academiaXidSep25/spring/springBatchV2

# Verificar que el archivo CSV está presente
ls -la src/main/resources/customers.csv

# Limpiar e instalar dependencias
mvn clean install

# Compilar el proyecto
mvn compile
```

11. Ejecución de la Aplicación

Ejecutar desde Maven

```
mvn spring-boot:run
```

Resultado esperado:

```
Started BatchProcessingDemoApplication in 4.567 seconds (JVM running for 5.234)
Application started on port 9191
```

Verificación: - La aplicación corre en puerto **9191** - Spring Batch crea automáticamente tablas de metadatos en MySQL: - `BATCH_JOB_EXECUTION` - `BATCH_JOB_EXECUTION_PARAMS` - `BATCH_STEP_EXECUTION` - `BATCH_JOB_INSTANCE` - Se crea la tabla `CUSTOMERS_INFO` (vacía inicialmente)

12. Ejecutar el Proceso Batch

Usar Postman o cURL para disparar el job:

```
curl -X POST http://localhost:9191/jobs/importCustomers
```

Logs esperados en consola:

```
Step: [csv-step] executed in 3.13 seconds
Step: [second-step] executed in 1.25 seconds
Job: [importCustomers] completed with status=COMPLETED in 4.38 seconds
```

13. Verificar Resultados

Verificar Step 1 (CSV → MySQL)

```
USE javatechie;
SELECT COUNT(*) FROM CUSTOMERS_INFO;
-- Resultado esperado: ~25 registros (solo mujeres de China)

SELECT * FROM CUSTOMERS_INFO LIMIT 5;
-- Todos los registros deben tener GENDER='Female' y COUNTRY='China'
```

Verificar Step 2 (MySQL → MongoDB)

```
use empleado
db.customers.count()
// Resultado esperado: ~25 registros

db.customers.find().limit(5)
// Todos los emails deben terminar en "@china.com"
```

Verificar Metadatos de Spring Batch

```
-- Ver historial de jobs
SELECT * FROM BATCH_JOB_EXECUTION
ORDER BY CREATE_TIME DESC;

-- Ver detalles de steps
SELECT step_name, status, read_count, write_count, commit_count
FROM BATCH_STEP_EXECUTION
ORDER BY CREATE_TIME DESC;
```

14. Rendimiento y Optimización

Sin Task Executor (Procesamiento Síncrono)

- **Tiempo**: ~6 segundos para 1000 registros
- **Características**: Los registros se procesan secuencialmente (ID: 1, 2, 3, 4…)

Con Task Executor (Procesamiento Paralelo - 10 hilos)

- **Tiempo**: ~2-3 segundos para 1000 registros
- **Características**: Los registros se procesan en paralelo (ID: 788, 366, 192… orden aleatorio)

¿Por qué el orden cambia? Cuando usas Task Executor, **10 hilos ejecutan concurrentemente**. No sabemos qué hilo procesará qué registro, por eso los IDs aparecen desordenados. Esto es **normal y esperado** en procesamiento paralelo.

15. Personalizar Filtros en ItemProcessor

Ejemplo: Filtrar solo clientes de Estados Unidos

```
// En CustomerProcessor.java
@Override
public Customer process(Customer customer) throws Exception {
    if(customer.getCountry().equals("United States")) {
        return customer;  // Procesar solo si es de Estados Unidos
    } else {
        return null;      // Filtrar fuera el resto
    }
}
```

Resultados después del cambio:

- **Registros procesados**: ~150 (solo de Estados Unidos)
- **Tiempo de ejecución**: ~1.5 segundos (menos registros = más rápido)

16. Monitoreo y Troubleshooting

Problemas Comunes y Soluciones

1. **Error: "Required a bean of type Job"**

   `Solución: Agregar @Bean a la definición del Job en SpringBatchConfig.java`

2. **Error: "Invalid setter method"**

   `Solución: Agregar @Data, @AllArgsConstructor, @NoArgsConstructor a las entidades`

3. **Error de conexión MySQL**

```
# Verificar que MySQL esté corriendo
sudo systemctl status mysql

# Verificar credenciales en application.properties
spring.datasource.username=root
spring.datasource.password=xideral1234
```

4. **Error de conexión MongoDB**

```
# Verificar que MongoDB esté corriendo con autenticación
mongo admin -u admin -p xideral4321
```

17. Comandos de Verificación

```
# Ver logs de la aplicación
tail -f logs/application.log

# Verificar procesos Java en ejecución
jps -l

# Verificar que el puerto 9191 está en uso
netstat -an | grep 9191

# Parar la aplicación
kill [PID_DEL_PROCESO]
```

Resumen del Flujo Completo

Lo que hace este proyecto:

1. **Preparación**:
   - Spring Boot 2.6.7 con Spring Batch habilitado
   - Conexiones a MySQL y MongoDB configuradas
   - Archivo CSV con 1000+ registros de clientes
2. **Step 1 - Procesamiento CSV → MySQL**:
   - Lee `customers.csv` línea por línea
   - Filtra solo **mujeres de China** usando `CustomerProcessor`
   - Guarda ~25 registros en MySQL tabla `CUSTOMERS_INFO`
   - Procesamiento paralelo con 10 hilos (2-3 segundos)
3. **Step 2 - Migración MySQL → MongoDB**:
   - Lee todos los registros de MySQL
   - Transforma emails agregando dominio **"@china.com"** usando `EmailDomainProcessor`
   - Guarda en MongoDB colección `customers`
4. **Resultado Final**:
   - **MySQL**: 25 mujeres de China con emails originales
   - **MongoDB**: Los mismos 25 registros con emails transformados a "@china.com"

Beneficios Demostrados:

- **Velocidad**: 1000 registros procesados en 2-3 segundos (vs 6+ segundos sin optimización)
- **Filtrado**: Solo procesa datos relevantes (mujeres de China)
- **Transformación**: Cambia emails automáticamente durante migración
- **Escalabilidad**: Procesamiento paralelo con múltiples hilos
- **Robustez**: Spring Batch maneja errores y reintentos automáticamente
- **Monitoreo**: Tablas de metadatos para tracking de jobs y steps

Conceptos Avanzados (Próximos Pasos)

Spring Batch Partitioning

El tutorial menciona **Spring Batch Partitioning** como el siguiente nivel: - Control total sobre qué hilo procesa qué registros - Ejemplo: Hilo 1 procesa registros 1-100, Hilo 2 procesa 101-200, etc. - Mayor control y predictibilidad en el procesamiento

Otros ItemWriters Disponibles

Además de `RepositoryItemWriter`, Spring Batch ofrece: - `JdbcBatchItemWriter` - Para operaciones JDBC optimizadas - `FlatFileItemWriter` - Para escribir a archivos CSV/TXT - `JsonFileItemWriter` - Para archivos JSON - `KafkaItemWriter` - Para Apache Kafka - Custom Writers para APIs REST, etc.

Configuraciones Adicionales

```
// Configurar chunk size dinámicamente
.chunk(100)  // Procesa 100 registros por transacción

// Configurar retry policy
.faultTolerant()
.retryLimit(3)
.retry(Exception.class)

// Skip registros problemáticos
.skipLimit(10)
.skip(Exception.class)
```

Conclusión

Este proyecto demuestra cómo **Spring Batch puede transformar el procesamiento de datos** de una tarea manual y lenta, a un proceso automatizado, rápido y confiable.

Antes (procesamiento manual): - Insertar 1000 registros uno por uno - Sin filtros automáticos - Sin transformaciones - Proceso lento y propenso a errores

Después (con Spring Batch): - Procesamiento automático de 1000+ registros - Filtros inteligentes (solo mujeres de China) - Transformaciones automáticas (emails a @china.com) - Migración entre bases de datos (MySQL → MongoDB) - Procesamiento paralelo (2-3 segundos) - Monitoreo y logging automático

¿El resultado? Un sistema ETL completo, escalable y profesional que puede manejar millones de registros con mínimo esfuerzo humano.