Spring Batch example

spring batch example from scratch

Here's a **complete step-by-step Spring Batch example from scratch**, including key concepts like `Job`, `Step`, `ItemReader`, `ItemProcessor`, and `ItemWriter`.

🔧 Goal

Let's build a **Spring Boot Batch application** that reads data from a **CSV file**, processes it (converts names to uppercase), and writes it to a **database (H2)**.

🧱 Project Structure

```css
spring-batch-demo/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com.example.batch/
│   │   │       ├── BatchConfig.java
│   │   │       ├── SpringBatchDemoApplication.java
│   │   │       ├── model/Person.java
│   │   │       ├── processor/PersonItemProcessor.java
│   │   │       └── repository/PersonRepository.java
│   │   └── resources/
│   │       ├── input.csv
│   │       └── application.yml
├── pom.xml
```

📦 1. `pom.xml`

```xml
```

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>spring-batch-demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <properties>
    <java.version>17</java.version>
    <spring.boot.version>3.1.0</spring.boot.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-batch</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

📁 2. Model: `Person.java`

```java
package com.example.batch.model;

import jakarta.persistence.*;
import lombok.*;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String firstName;
    private String lastName;
}
```

📂 3. Repository: `PersonRepository.java`

```java
package com.example.batch.repository;

import com.example.batch.model.Person;
import org.springframework.data.jpa.repository.JpaRepository;

public interface PersonRepository extends JpaRepository<Person, Long> {}
```

⚙️ 4. Processor: `PersonItemProcessor.java`

```java
```

```
package com.example.batch.processor;

import com.example.batch.model.Person;
import org.springframework.batch.item.ItemProcessor;

public class PersonItemProcessor implements ItemProcessor<Person, Person> {
    @Override
    public Person process(Person person) {
        person.setFirstName(person.getFirstName().toUpperCase());
        person.setLastName(person.getLastName().toUpperCase());
        return person;
    }
}
```

🛠️ 5. Batch Config: `BatchConfig.java`

```java
package com.example.batch;

import com.example.batch.model.Person;
import com.example.batch.processor.PersonItemProcessor;
import com.example.batch.repository.PersonRepository;
import org.springframework.batch.core.*;
import org.springframework.batch.core.configuration.annotation.*;
import org.springframework.batch.core.launch.support.RunIdIncrementer;
import org.springframework.batch.item.data.RepositoryItemWriter;
import org.springframework.batch.item.file.*;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.batch.item.file.mapping.*;
import org.springframework.batch.item.file.transform.DelimitedLineTokenizer;
import org.springframework.beans.factory.annotation.*;
import org.springframework.context.annotation.*;
import org.springframework.core.io.ClassPathResource;

@Configuration
@EnableBatchProcessing
public class BatchConfig {

    @Bean
```

```java
  public FlatFileItemReader<Person> reader() {
    return new FlatFileItemReaderBuilder<Person>()
        .name("personItemReader")
        .resource(new ClassPathResource("input.csv"))
        .delimited()
        .names("firstName", "lastName")
        .fieldSetMapper(new BeanWrapperFieldSetMapper<>() {{
          setTargetType(Person.class);
        }})
        .build();
  }

  @Bean
  public PersonItemProcessor processor() {
    return new PersonItemProcessor();
  }

  @Bean
  public RepositoryItemWriter<Person> writer(PersonRepository repository) {
    RepositoryItemWriter<Person> writer = new RepositoryItemWriter<>();
    writer.setRepository(repository);
    writer.setMethodName("save");
    return writer;
  }

  @Bean
  public Job importUserJob(JobRepository jobRepository, Step step1) {
    return new JobBuilder("importUserJob", jobRepository)
        .incrementer(new RunIdIncrementer())
        .flow(step1)
        .end()
        .build();
  }

  @Bean
  public Step step1(JobRepository jobRepository, PlatformTransactionManager transactionManager) {
    return new StepBuilder("step1", jobRepository)
        .<Person, Person>chunk(10, transactionManager)
        .reader(reader())
        .processor(processor())
        .writer(writer(null))
```

```
                .build();
    }
}
```

🚀 6. Main Class: `SpringBatchDemoApplication.java`

java

```java
package com.example.batch;

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

@SpringBootApplication
public class SpringBatchDemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBatchDemoApplication.class, args);
    }
}
```

📝 7. Input CSV File: `src/main/resources/input.csv`

```
firstName,lastName
john,doe
jane,smith
alex,jones
```

⚙️ 8. Configuration: `application.yml`

yaml

```yaml
spring:
  datasource:
    url: jdbc:h2:mem:testdb
    driver-class-name: org.h2.Driver
    username: sa
    password:
  h2:
    console:
      enabled: true
  jpa:
    hibernate:
      ddl-auto: create
    show-sql: true
    properties:
      hibernate.format_sql: true

logging:
  level:
    org.springframework.batch: INFO
```

✅ How to Run

1. `mvn spring-boot:run`
2. Open H2 Console: http://localhost:8080/h2-console
   - JDBC URL: `jdbc:h2:mem:testdb`
3. Check `PERSON` table. You'll see names in **UPPERCASE**.

🧠 Key Spring Batch Concepts

Concept	Explanation
Job	Represents the batch job (one or more steps).
Step	A phase in the job (e.g., read → process → write).
ItemReader	Reads input data (CSV, DB, etc).
ItemProcessor	Business logic to transform data.
ItemWriter	Writes data to a destination (DB, file).

Concept	Explanation
JobRepository	Stores job execution metadata.
Chunk	A set of records processed together for performance.

Do you want this job to be triggered via REST endpoint too? Or schedule it with `@Scheduled`?

explain Key Features and Concepts:

Job Management:

Step:

Chunk Processing:

ItemReader, ItemProcessor, ItemWriter:

Transaction Management:

Error Handling:

Job Repository:

explain with code

Here's a **detailed explanation** of key Spring Batch features and concepts — with **real code snippets** from the Spring Batch project you just built (CSV → DB processor):

✅ 1. Job Management

A **Job** is the **container for steps**. It defines the entire batch process.

🔹 Code

java

```java
@Bean
public Job importUserJob(JobRepository jobRepository, Step step1) {
    return new JobBuilder("importUserJob", jobRepository)
            .incrementer(new RunIdIncrementer()) // allows running the same job multiple times
            .flow(step1)
            .end()
            .build();
}
```

💡 Notes

- A `Job` can consist of **multiple steps**.
- `RunIdIncrementer` ensures **unique job instance ID** per run (for reruns).

✅ 2. Step

Each **Step** represents an **independent phase** in the job (e.g., read-process-write).

🔹 Code

```java
@Bean
public Step step1(JobRepository jobRepository, PlatformTransactionManager
transactionManager) {
    return new StepBuilder("step1", jobRepository)
            .<Person, Person>chunk(10, transactionManager)
            .reader(reader())
            .processor(processor())
            .writer(writer(null))
            .build();
}
```

💡 Notes

- A **Step** can be `Tasklet` based (single task) or `Chunk` based (loop over items).
- It's executed **sequentially** in the job.

✅ 3. Chunk Processing

Spring Batch reads and processes data in **chunks** (batches), improving performance and memory usage.

```java
.chunk(10, transactionManager)
```

🔹 What happens:

- Read up to **10 records** (or fewer, if end of input).
- Process each.
- Write them **together** as a transaction.
- Repeat until the end.

✅ 4. ItemReader, ItemProcessor, ItemWriter

These are the **core interfaces** of chunk-based processing.

🔹 `ItemReader<T>`

Reads **one item** at a time from the input (file, DB, etc).

```java
@Bean
public FlatFileItemReader<Person> reader() {
    return new FlatFileItemReaderBuilder<Person>()
            .name("personItemReader")
            .resource(new ClassPathResource("input.csv"))
            .delimited()
            .names("firstName", "lastName")
            .fieldSetMapper(new BeanWrapperFieldSetMapper<>() {{
                setTargetType(Person.class);
            }})
            .build();
}
```

🔹 `ItemProcessor<I, O>`

Processes and transforms the data.

```java
public class PersonItemProcessor implements ItemProcessor<Person, Person> {
  @Override
  public Person process(Person person) {
    person.setFirstName(person.getFirstName().toUpperCase());
    person.setLastName(person.getLastName().toUpperCase());
    return person;
  }
}
```

🔹 `ItemWriter<T>`

Writes the **processed items** to the destination (DB, file, etc).

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

✅ 5. Transaction Management

Each **chunk** is a **transactional unit**. If any error occurs while reading/processing/writing, the whole chunk is rolled back.

🔹 Code

```java
.chunk(10, transactionManager)
```

Spring Batch uses the provided `PlatformTransactionManager` to:

- **Begin** a transaction before reading the chunk.

- **Commit** the transaction if processing/writing succeeds.
- **Rollback** if any error occurs in the chunk.

✅ 6. Error Handling

Spring Batch provides built-in **fault tolerance**:

🔹 Skip and Retry (optional):

```java
@Bean
public Step step1(...) {
    return new StepBuilder("step1", jobRepository)
            .<Person, Person>chunk(10, transactionManager)
            .reader(reader())
            .processor(processor())
            .writer(writer(null))
            .faultTolerant()
            .skipLimit(5)
            .skip(FlatFileParseException.class)
            .build();
}
```

💡 You can:

- `skip()` certain exceptions.
- `retry()` a specific number of times.
- Set `skipLimit` to prevent job from continuing endlessly.

✅ 7. Job Repository

Spring Batch stores all **job metadata** (job instances, executions, step executions, statuses) in the **JobRepository**.

- Automatically created if `@EnableBatchProcessing` is present.
- Uses DB tables (`BATCH_JOB_INSTANCE` , `BATCH_JOB_EXECUTION` , etc.)

🔹 Example:

```sql
```

```
SELECT * FROM BATCH_JOB_EXECUTION;
```

You'll see:

- Start time
- End time
- Status: `COMPLETED`, `FAILED`, etc.

✅ Summary Table

Feature	Description
Job	A container that defines the entire batch process
Step	A phase in a job (e.g., read-process-write)
Chunk Processing	Divides work into chunks; each chunk is a transaction
Reader	Reads one item from input source
Processor	Transforms input into output
Writer	Writes processed items to output
Transaction Mgmt	Each chunk is transactional (commit/rollback)
Error Handling	Supports retry, skip, and listeners for failure recovery
Job Repository	Tracks job/step execution status and metadata

Would you like me to show:

- How to schedule this job (`@Scheduled`)?
- Or expose it through a REST API?

Let me know!

here i need complete example from scratch one db data copy to another db for everyday morning 9 am

data bases are postgressdb to mysql db

Here’s a **complete Spring Batch example from scratch** that:

- ✅ **Reads data from a PostgreSQL database**
- ✅ **Processes (or copies directly)**
- ✅ **Writes to a MySQL database**
- ✅ **Runs every day at 9 AM**

✅ Use Case

Copy data from PostgreSQL to MySQL every day at 9 AM, using Spring Boot + Spring Batch + `@Scheduled`.

📦 Project Dependencies

`pom.xml`

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>batch-db-copy</artifactId>
  <version>1.0.0</version>

  <properties>
    <java.version>17</java.version>
    <spring-boot.version>3.1.0</spring-boot.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-batch</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- PostgreSQL -->
    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
    </dependency>

    <!-- MySQL -->
    <dependency>
      <groupId>com.mysql</groupId>
      <artifactId>mysql-connector-j</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

🏗️ Database Table (Common for both DBs)

sql

```
CREATE TABLE person (
  id SERIAL PRIMARY KEY,
  first_name VARCHAR(50),
  last_name VARCHAR(50)
);
```

🗃️ `application.yml`

yaml

```yaml
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/targetdb
    username: root
    password: root
    driver-class-name: com.mysql.cj.jdbc.Driver

  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true

  batch:
    jdbc:
      initialize-schema: always

source:
  datasource:
    url: jdbc:postgresql://localhost:5432/sourcedb
    username: postgres
    password: postgres
    driver-class-name: org.postgresql.Driver
```

🧱 `Person.java`

java

```java
package com.example.batch.model;

import jakarta.persistence.*;
import lombok.*;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Person {
    @Id
    private Long id;

    private String firstName;
    private String lastName;
}
```

🔄 DataSourceConfig.java (Source DB)

java

```java
package com.example.batch.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.jdbc.DataSourceBuilder;
import org.springframework.context.annotation.*;

import javax.sql.DataSource;

@Configuration
public class SourceDataSourceConfig {

    @Bean(name = "sourceDataSource")
    @Primary
    @ConfigurationProperties(prefix = "source.datasource")
    public DataSource sourceDataSource() {
        return DataSourceBuilder.create().build();
    }
}
```

📁 `PersonRepository.java`

```java
package com.example.batch.repository;

import com.example.batch.model.Person;
import org.springframework.data.jpa.repository.JpaRepository;

public interface PersonRepository extends JpaRepository<Person, Long> {}
```

⚙️ `BatchConfig.java`

```java
package com.example.batch.config;

import com.example.batch.model.Person;
import com.example.batch.repository.PersonRepository;
import org.springframework.batch.core.*;
import org.springframework.batch.core.configuration.annotation.*;
import org.springframework.batch.core.launch.support.RunIdIncrementer;
import org.springframework.batch.item.database.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.context.annotation.*;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.scheduling.annotation.EnableScheduling;

import javax.sql.DataSource;

@Configuration
@EnableBatchProcessing
@EnableScheduling
public class BatchConfig {

    @Autowired
    private PersonRepository personRepository;

    @Bean
    public JdbcCursorItemReader<Person> reader(@Qualifier("sourceDataSource")
DataSource sourceDataSource) {
        JdbcCursorItemReader<Person> reader = new JdbcCursorItemReader<>();
        reader.setDataSource(sourceDataSource);
        reader.setSql("SELECT id, first_name, last_name FROM person");
        reader.setRowMapper(new BeanPropertyRowMapper<>(Person.class));
        return reader;
    }

    @Bean
    public ItemWriter<Person> writer() {
        return persons -> personRepository.saveAll(persons);
    }

    @Bean
    public Step step(JobRepository jobRepository, PlatformTransactionManager
transactionManager,
                     JdbcCursorItemReader<Person> reader, ItemWriter<Person> writer) {
        return new StepBuilder("copyStep", jobRepository)
                .<Person, Person>chunk(10, transactionManager)
                .reader(reader)
                .writer(writer)
                .build();
    }

    @Bean
    public Job copyJob(JobRepository jobRepository, Step step) {
        return new JobBuilder("copyJob", jobRepository)
                .incrementer(new RunIdIncrementer())
                .flow(step)
                .end()
                .build();
    }
}
```

🕘 `JobScheduler.java` (Scheduled to run at 9 AM)

```java
package com.example.batch.scheduler;

import org.springframework.batch.core.*;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.scheduling.annotation.*;
import org.springframework.stereotype.Component;

import java.util.Date;

@Component
public class JobScheduler {

    @Autowired
    private JobLauncher jobLauncher;

    @Autowired
    private Job copyJob;

    // Run every day at 9:00 AM
    @Scheduled(cron = "0 0 9 * * ?")
    public void runJob() throws Exception {
        JobParameters params = new JobParametersBuilder()
                .addLong("time", System.currentTimeMillis()) // ensure uniqueness
                .toJobParameters();

        jobLauncher.run(copyJob, params);
    }
}
```

🚀 `SpringBatchDbCopyApplication.java`

```java

package com.example.batch;

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

@SpringBootApplication
public class SpringBatchDbCopyApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBatchDbCopyApplication.class, args);
    }
}
```

✅ Final Notes

Feature	Detail
Read DB	PostgreSQL (`sourceDataSource`)
Write DB	MySQL (default Spring `datasource`)
Scheduler	Every day at **9 AM** (cron: `0 0 9 * * ?`)
Frameworks	Spring Boot + Spring Batch + Spring Data JPA
Tables Needed	Both DBs should have the same `person` table schema

🧪 Test the Job

1. Insert data into **PostgreSQL** `person` table.
2. Run app or wait till 9 AM.
3. Data will appear in **MySQL** `person` table.

Would you like:

- To include logging and error handling?
- To export job status to a table or email?

Let me know, and I'll add it.