Spring Batch

Why large-scale batch processing needs a structure `@Scheduled` alone can't provide -- the Job/Step/chunk-oriented processing mental model, ItemReader/ItemProcessor /ItemWriter through a single CSV-to-database order import example, the JobParameters /JobInstance/JobExecution distinction, restartability via the JobRepository, skip/retry fault tolerance, and practical integration with `@Scheduled`. The 4th lesson in the Advanced Spring series.

Advanced 50 min

Last updated: August 24, 2026
Read online: https://www.learnforgex.com/en/topics/spring-batch

---

"Task Execution & Scheduling" answered "when should this work start, and how do I run it without blocking a caller?" This lesson answers a different question entirely: once a piece of work is genuinely large — hundreds of thousands of records, not a handful — how do you structure, execute, track, and recover it? Spring Batch is Spring's answer, and it's built on top of neither `@Async` nor `@Scheduled`; it's a separate concern that often gets triggered BY one of them.

The Real Problem: A Nightly Batch of 500,000 Records

Picture a real requirement: every night at 2 AM, the application must process 500,000 customer records — read them from the database, validate and transform each one, write the results somewhere else, survive a failure partway through without corrupting anything, and be able to restart from where it failed instead of reprocessing all 500,000 records from scratch.

The naive approach looks obvious at first:

```
@Scheduled(cron = "0 0 2 * * *")
public void processCustomers() {
    // read 500,000 records
    // process them
    // write them
}
```

This compiles, and for a few hundred rows it might even work. At real scale it falls apart: loading 500,000 records into memory at once is wasteful (or impossible); a single giant transaction around all of it either locks resources for far too long or, if it's not transactional at all, leaves the database in a half-written state on any failure; and if the process crashes

at record 300,001, there is no record anywhere of how far it actually got — a restart means starting over from record 1.

`@Scheduled` and Spring Batch are not solving the same problem — don't reach for one expecting the other's guarantees.

```
@Scheduled
    → "WHEN should this work start?"

Spring Batch
    → "HOW should a large batch operation be structured,
       executed, tracked, and restarted?"
```

The two are not competitors — they're often used together, each doing only the part it's actually good at:

```
@Scheduled
     |
     v
launch Spring Batch Job
     |
     v
Spring Batch manages the actual batch processing
```

A Mental Model: Job, Step, and Chunk-Oriented Processing

Before any Spring Batch class or annotation, here is the shape everything else in this lesson fills in:

```
Job
 |
 +-- Step 1
 |     |
 |     +-- read
 |     +-- process
 |     +-- write
 |
 +-- Step 2
       |
       +-- read
       +-- process
       +-- write
```

A **Job** is the overall batch process — "import tonight's orders," end to end. A **Step** is one phase of that process — a Job with several genuinely distinct phases (import, then reconcile, then report) would have several Steps, run in sequence. A Step is commonly **chunk-oriented**: it reads a batch of items, optionally transforms each one, and writes them — and inside a chunk-oriented step, three roles do the actual work: an `ItemReader` reads items one at a time, an `ItemProcessor` optionally transforms or validates each one, and an `ItemWriter`

writes a whole group of them at once. Spring Batch itself manages the loop that drives these three — you write what each one does, not the loop that calls them.

The Running Example: Importing Orders from a CSV File

Every piece of this lesson builds toward ONE concrete scenario, reused throughout rather than a new toy example per section: **importing customer orders from a CSV file into a database.**

```
orderId,customerId,amount
1001,C001,125.50
1002,C002,89.90
1003,C003,250.00
```

The job reads each row from `orders.csv`, converts it into an `Order`, validates or transforms it, and writes valid orders to the database — exactly the reader/processor/writer roles from the mental model above, applied to a real pipeline:

```
orders.csv
    |
    v
ItemReader
    |
    v
Order
    |
    v
ItemProcessor
    |
    v
Processed Order
    |
    v
ItemWriter
    |
    v
Database
```

Chunk Processing: Reading, Writing, and Committing in Groups

`.chunk(100)` is easy to read as "process 100 items" and move on — but what actually happens underneath is the single most important mechanic in this lesson.

```
Read item 1
Read item 2
Read item 3
...
Read item 100
        |
        v
Process items
        |
        v
Write items 1-100
        |
        v
COMMIT TRANSACTION
```

Then the exact same cycle repeats, chunk after chunk:

```
101-200
        |
        v
write
        |
        v
COMMIT

201-300
        |
        v
write
        |
        v
COMMIT
```

Chunking matters for several concrete reasons at once: memory usage stays bounded (100 `Order` objects in memory, not 500,000); each chunk is its own transaction boundary, so a failure doesn't leave a half-written mess spanning the entire job; and it's what makes restart meaningful at all — without chunk boundaries, there would be no smaller unit than "the whole job" to know the status of.

```
Items 1-100     → committed
Items 101-200   → committed
Items 201-300   → FAILURE
```

Conceptually, the first 200 items are safely committed, and only the third chunk failed — Spring Batch's execution metadata (covered shortly) is what lets the framework know processing reached chunk 3, not chunk 1. Exactly which items get reprocessed on a restart depends on the job's configuration and the state that was persisted — a detail this lesson comes back to in "Restartability and the JobRepository," rather than promised here in the abstract.

A Minimal, Complete Job Configuration

With the mental model in place, here is a genuinely complete (if minimal) Job and Step for the order-import example.

```
import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.job.builder.JobBuilder;
import org.springframework.batch.core.repository.JobRepository;
import org.springframework.batch.core.step.builder.StepBuilder;
import org.springframework.batch.item.ItemProcessor;
import org.springframework.batch.item.ItemReader;
import org.springframework.batch.item.ItemWriter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.transaction.PlatformTransactionManager;

record Order(String orderId, String customerId, java.math.BigDecimal amount) {
}

// The two building blocks every Spring Batch job needs: a Job (the whole
// process) built from one or more Steps (a phase of that process). Both
// builders require a JobRepository -- Spring Batch's own store for
// execution state, covered later in this lesson -- because every Job and
// Step it creates is TRACKED there from the moment it runs, not just
// executed and forgotten.
@Configuration
class OrderImportJobConfig {

    @Bean
    public Job orderImportJob(JobRepository jobRepository, Step importStep) {
        return new JobBuilder("orderImportJob", jobRepository)
                .start(importStep)
                .build();
    }

    // chunk(100, transactionManager) is what makes this step CHUNK-ORIENTED:
    // reader/processor/writer are wired together here, and Spring Batch
    // itself drives the read-process-write loop between them in groups of
    // 100 -- see "Chunk Processing" for exactly what that means.
    @Bean
    public Step importStep(JobRepository jobRepository,
                           PlatformTransactionManager transactionManager,
                           ItemReader<Order> orderItemReader,
                           ItemProcessor<Order, Order> orderItemProcessor,
                           ItemWriter<Order> orderItemWriter) {

        return new StepBuilder("importStep", jobRepository)
                .<Order, Order>chunk(100, transactionManager)
                .reader(orderItemReader)
                .processor(orderItemProcessor)
                .writer(orderItemWriter)
                .build();
    }
}
```

Read it top to bottom the way Spring Batch itself executes it:

```
Job
 ↓
start(importStep)
 ↓
Step
 ↓
chunk(100)
 ↓
reader → processor → writer
```

`orderImportJob(...)` is the Job — here, a single Step (`importStep`) is its entire process; a `JobBuilder` needs a `JobRepository` because, as covered further below, every Job Spring Batch creates is tracked there. `importStep(...)` is where the real work is described: `.chunk(100, transactionManager)` declares chunk-oriented processing with a chunk size of 100, and `.reader(...)` / `.processor(...)` / `.writer(...)` plug in the three roles from the mental model — each supplied as its own bean, detailed next.

ItemReader, ItemProcessor, and ItemWriter for the Order Import

Each of the three roles referenced above needs a concrete implementation for THIS example — reading CSV rows, filtering invalid amounts, and writing to the database.

```
import org.springframework.batch.item.ItemProcessor;
import org.springframework.batch.item.database.JdbcBatchItemWriter;
import org.springframework.batch.item.database.builder.JdbcBatchItemWriterBuilder;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.ClassPathResource;

import javax.sql.DataSource;
import java.math.BigDecimal;

record Order(String orderId, String customerId, BigDecimal amount) {
}

// The three pieces a chunk-oriented step actually drives: read one Order
// at a time from orders.csv, optionally transform/validate it, then write
// it. Spring Batch calls these in a loop -- reader.read() repeatedly until
// the chunk size is reached, processor.process() on each item, then a
// SINGLE writer.write(...) call for the whole chunk -- see "Chunk
// Processing" for why that grouping matters.
@Configuration
class OrderImportComponents {

    @Bean
    public FlatFileItemReader<Order> orderItemReader() {
        return new FlatFileItemReaderBuilder<Order>()
```

```
                .name("orderItemReader")
                .resource(new ClassPathResource("orders.csv"))
                .linesToSkip(1) // skip the CSV header row (orderId,customerId,amount)
                .delimited()
                .names("orderId", "customerId", "amount")
                .fieldSetMapper(fieldSet -> new Order(
                        fieldSet.readString("orderId"),
                        fieldSet.readString("customerId"),
                        fieldSet.readBigDecimal("amount")))
                .build();
    }

    // Returning null here is a deliberate Spring Batch convention: it
    // means "filter this item out." A filtered item is silently dropped
    // and never reaches the writer -- this is NOT an error and does not
    // fail the step, unlike the validation failures covered later in
    // "Fault Tolerance: Skip and Retry."
    @Bean
    public ItemProcessor<Order, Order> orderItemProcessor() {
        return order -> {
            if (order.amount().signum() < 0) {
                return null; // filtered out: a negative amount is not written, not an error
            }
            return order;
        };
    }

    @Bean
    public JdbcBatchItemWriter<Order> orderItemWriter(DataSource dataSource) {
        return new JdbcBatchItemWriterBuilder<Order>()
                .dataSource(dataSource)
                .sql("INSERT INTO orders (order_id, customer_id, amount) VALUES (:orderId, :
customerId, :amount)")
                .beanMapped()
                .build();
    }
}
```

`orderItemReader()` builds a `FlatFileItemReader<Order>` that reads `orders.csv`, skips its header row, splits each line on the delimiter, and maps the three columns onto an `Order`. `orderItemProcessor()` shows an important, easy-to-miss Spring Batch behavior: **returning `null` from an `ItemProcessor` filters the item out** — it is silently dropped and never reaches the writer, which is exactly how a negative-amount order gets excluded here without failing anything. `orderItemWriter()` writes each chunk's `Order`s to the database with a single batched SQL insert per chunk, rather than one insert per row.

> Filtering (returning `null`) and failing (throwing an exception) are different outcomes with different consequences — filtering quietly excludes one item and the step continues normally; throwing signals a real problem, and by default fails the whole step. "Fault Tolerance: Skip and Retry," later in this lesson, covers how to make certain exceptions recoverable instead.

JobParameters, JobInstance, and JobExecution

Running the SAME logical job again isn't automatically the same thing as running it for the first time — Spring Batch needs a way to tell these apart, and that's what these three concepts are for.

```
Job: orderImportJob
JobParameters:
    file=orders.csv
    businessDate=2026-08-24
```

A `Job` combined with a specific set of `JobParameters` identifies a `JobInstance`:

```
Job
+
JobParameters
        |
        v
JobInstance
```

This matters because a batch job is usually about a specific input, not just "run the logic again." Two different `businessDate` values genuinely represent two different pieces of work:

```
businessDate=2026-08-24
        → one logical job instance

businessDate=2026-08-25
        → another logical job instance
```

But even a single `JobInstance` can be ATTEMPTED more than once — if it fails and is restarted, that's still the same logical instance, just a new attempt at it. That attempt is a `JobExecution`:

```
Job
 |
 +-- JobInstance (businessDate=2026-08-24)
       |
       +-- JobExecution #1 → FAILED
       |
       +-- JobExecution #2 → COMPLETED
```

A `StepExecution` is the same idea one level down — a single Step's own attempt, within a particular `JobExecution`. Spring Batch stores all of this (which you'll see next) precisely so a failed `JobExecution` can be told apart from a successful one for the exact same `JobInstance`, instead of every run looking identical.

Restartability and the JobRepository

The reason all of the bookkeeping above exists becomes concrete with a failure scenario.

```
500,000 records

1-100,000       ✓
100,001-200,000 ✓
200,001-300,000 ✓
300,001-400,000 ✗ application crashes
```

A serious batch framework shouldn't have to treat this as "the whole operation failed, start over" — it should be able to persist enough execution state to support restarting the failed attempt, picking up from roughly where it left off, rather than blindly reprocessing the first 300,000 records that already succeeded. This is exactly what the `JobRepository` is for:

```
Your batch job
      |
      v
JobRepository
      |
      +-- job execution status
      +-- step execution status
      +-- execution metadata
      +-- restart-related state
```

This is what separates Spring Batch from simply writing:

```
while (...) {
    read();
    process();
    write();
}
```

A plain loop like that has no memory of its own progress — if it dies at record 300,001, nothing anywhere records that fact. The `JobRepository` is where `JobExecution`s and `StepExecution`s actually get persisted, along with an `ExecutionContext` — a small bag of state a Step can use to remember details relevant to resuming it. This lesson introduces `ExecutionContext` only far enough to explain why restart is possible at all; exactly which items get skipped versus reprocessed on a given restart depends on the step's own configuration, not on some universal guarantee.

What Happens When a Step Fails

Put concretely, a chunk-oriented step's failure path looks like this:

```
Reader
   ↓
Processor
   ↓
Writer
   ↓
Database failure
```

A few things are true regardless of exactly where in a chunk something goes wrong: the chunk is the transaction boundary, so a failure inside it rolls back that entire chunk's writes, not just the one item that triggered it; the `JobExecution` / `StepExecution` metadata records that this attempt failed and roughly how far it got; and whether — and how precisely — a restart resumes from that point depends on the step's configuration (chunk size, whether it's restartable, what state it stores), not on an automatic, universal guarantee that every failure resumes from the exact failed item.

Fault Tolerance: Skip and Retry

Only after the core model — Job, Step, reader/processor/writer, chunks, the repository — makes sense does it make sense to introduce fault tolerance for individual bad items.

Picture 10,000 records where record #532 has malformed data. Without fault tolerance, that one bad record fails the ENTIRE step. With it configured:

```
import org.springframework.batch.core.Step;
import org.springframework.batch.core.repository.JobRepository;
import org.springframework.batch.core.step.builder.StepBuilder;
import org.springframework.batch.item.ItemProcessor;
import org.springframework.batch.item.ItemReader;
import org.springframework.batch.item.ItemWriter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.dao.TransientDataAccessException;
import org.springframework.transaction.PlatformTransactionManager;

record Order(String orderId, String customerId, java.math.BigDecimal amount) {
}

class InvalidOrderException extends RuntimeException {
    InvalidOrderException(String message) {
        super(message);
    }
}

// The SAME importStep from "A Minimal, Complete Job Configuration", with
// fault tolerance added -- faultTolerant() switches on Spring Batch's
// skip/retry machinery for this step, which is OFF by default.
@Configuration
class FaultTolerantImportStepConfig {

    @Bean
    public Step importStep(JobRepository jobRepository,
                           PlatformTransactionManager transactionManager,
                           ItemReader<Order> orderItemReader,
                           ItemProcessor<Order, Order> orderItemProcessor,
                           ItemWriter<Order> orderItemWriter) {

        return new StepBuilder("importStep", jobRepository)
                .<Order, Order>chunk(100, transactionManager)
                .reader(orderItemReader)
                .processor(orderItemProcessor)
                .writer(orderItemWriter)
                .faultTolerant()
                // A genuinely malformed row (a missing customerId, say)
                // throws InvalidOrderException instead of being silently
                // filtered the way a negative amount is -- skip() lets the
                // STEP continue past it instead of failing the entire job,
                // up to 10 such rows.
                .skip(InvalidOrderException.class)
                .skipLimit(10)
                // A database hiccup is often TEMPORARY -- retry() tries the
                // same write again (up to 3 times) before giving up,
                // instead of treating a transient failure as permanent.
                .retry(TransientDataAccessException.class)
                .retryLimit(3)
                .build();
    }
}
```

`.faultTolerant()` switches this behavior on for the step — it's off by default. `.skip(InvalidOrderException.class).skipLimit(10)` means: if this specific kind of error occurs, don't fail the whole job immediately — skip that one item and continue, but only up to 10 skips total; the 11th failure of that type still fails the step. `.retry(TransientDataAccessException.class).retryLimit(3)` means something different: this kind of error may be temporary (a brief database hiccup), so retry the SAME operation up to 3 times before giving up on it — skip is about tolerating a genuinely bad item, retry is about tolerating a possibly-temporary failure of an otherwise-good one.

Combining @Scheduled with Spring Batch

This lesson follows "Task Execution & Scheduling" directly, so it's worth connecting the two mechanisms explicitly.

```
import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import java.time.LocalDate;

// @Scheduled decides WHEN this runs; everything after that line is Spring
// Batch deciding HOW the actual import is executed, tracked, and (if
// needed) restarted -- "Task Execution & Scheduling" and this lesson
// working together, each doing only the job it's actually responsible for.
@Component
class NightlyOrderImportScheduler {

    private final JobLauncher jobLauncher;
    private final Job orderImportJob;

    NightlyOrderImportScheduler(JobLauncher jobLauncher, Job orderImportJob) {
        this.jobLauncher = jobLauncher;
        this.orderImportJob = orderImportJob;
    }

    @Scheduled(cron = "0 0 2 * * *")
    public void launchNightlyImport() throws Exception {
        // JobParameters identify THIS run -- "file" says which CSV to
        // read, "businessDate" says which day's import this is. Together
        // with the Job itself, these parameters are what Spring Batch uses
        // to tell one logical run apart from another -- see "JobParameters,
        // JobInstance, and JobExecution".
        JobParameters jobParameters = new JobParametersBuilder()
                .addString("file", "orders.csv")
                .addLocalDate("businessDate", LocalDate.now())
                .toJobParameters();

        jobLauncher.run(orderImportJob, jobParameters);
    }
}
```

`@Scheduled(cron = "0 0 2 * * *")` decides WHEN `launchNightlyImport()` runs — nothing new from the previous lesson. What's new is what it does: it builds `JobParameters` (the `file` / `businessDate` pair from earlier) and hands them, along with the `Job` itself, to a `JobLauncher` — the object that actually starts a `JobInstance` / `JobExecution` running.

```
2:00 AM
   |
   v
@Scheduled
   |
   v
JobLauncher
   |
   v
Spring Batch Job
   |
   v
Step
   |
   v
Reader → Processor → Writer
```

The distinction to hold onto: `@Scheduled` decides when to start; Spring Batch decides how to execute and manage the batch process once it does. This is one of the two central takeaways of this lesson.

@Async vs. @Scheduled vs. Spring Batch

With all three mechanisms from this category now covered, it's worth stating precisely what each one is actually for:

```
@Async
    → Run this method in the background.

@Scheduled
    → Start this method according to a schedule.

Spring Batch
    → Execute and manage a structured batch job,
      including steps, chunks, transactions,
      execution metadata, failure handling and restartability.
```

These aren't competing choices for the same problem — they coexist, and often do, exactly as `ScheduledOrderImportLauncher` shows: `@Scheduled` triggers a launch, which is itself neither `@Async` nor a batch job by itself, but starts one.

Putting It All Together: The Nightly Order Import, End to End

```
Nightly order import

@Scheduled
     |
     v
JobLauncher
     |
     v
Order Import Job
     |
     v
Import Step
     |
     +--> ItemReader
     |
     +--> ItemProcessor
     |
     +--> ItemWriter
     |
     v
Chunk transaction
     |
     v
JobRepository
```

In plain language: the scheduler triggers the job at 2 AM; Spring Batch creates or identifies the `JobExecution` for this `JobInstance`; the step starts; the reader reads `Order`s from the CSV; the processor validates and transforms each one (filtering some, as covered earlier); the writer writes the survivors to the database; all of this happens in chunks of 100; each chunk's writes commit together as one transaction; execution metadata is recorded in the `JobRepository` throughout; and if the job fails partway through, that stored execution state is what a restart can use to avoid starting completely over.

What Spring Batch Is NOT

A few explicit non-goals, to keep it from blurring together with the previous lesson: Spring Batch is not simply another scheduler — it has nothing to say about WHEN to run (that's `@Scheduled`'s job, or a manual trigger, or another scheduler entirely). It is not simply a thread pool — chunking and transactions are its core concern, not concurrent execution. It is not simply a `while` loop around database records — the `JobRepository`'s tracked execution state is what a plain loop never has. And it is not automatically a parallel-processing framework — a standard chunk-oriented step, exactly like the one built in this lesson, runs sequentially; genuine parallelism (partitioning, parallel steps) is a distinct, more advanced topic this lesson only names, not teaches.

Best Practices

- Reach for Spring Batch when the real requirement is structured, trackable, restartable processing of a large dataset — not for every scheduled task, which `@Scheduled` alone often already covers.
- Choose a chunk size deliberately: large enough to be efficient, small enough that a rollback or a restart doesn't waste excessive work.
- Use `ItemProcessor` returning `null` for legitimate filtering, and reserve thrown exceptions for genuine failures `skip` / `retry` should react to.
- Keep `skipLimit` / `retryLimit` values deliberately small and specific to the exception types that are genuinely tolerable — a bare `.skip(Exception.class)` hides real bugs behind a high skip count.
- Let `@Scheduled` decide when a job launches and let Spring Batch decide how it runs — don't blur the two together in one method.

Common Mistakes

- Assuming `@Scheduled` itself provides any of Spring Batch's tracking, chunking, or restart behavior — it only decides when something starts.
- Confusing a `Job` (the reusable definition) with a `JobExecution` (one attempt at running it) — the same `Job` bean produces many `JobExecution`s over time.
- Confusing a `JobInstance` (a `Job` plus its identifying `JobParameters`) with a `JobExecution` (one attempt at that specific instance) — one `JobInstance` can have several `JobExecution`s if it's retried.
- Not accounting for chunk boundaries — assuming every item is committed independently, when in fact a whole chunk commits (or rolls back) together.
- Assuming every failure automatically resumes from the exact failed item in every configuration, rather than depending on the step's own restart configuration.
- Reaching for partitioning or parallel steps before a plain, sequential chunk-oriented step is well understood.
- Creating several unrelated toy jobs to demonstrate different APIs instead of building depth in one coherent example, the way this lesson stuck to a single order-import job throughout.
- Writing the full Job/Step configuration before explaining the execution model it implements — the code means little without the mental model from earlier in this lesson.

Summary, Cheat Sheet, and Glossary

Summary

For every major abstraction in this lesson, the question it answers:

```
Job
→ What is the overall unit of work?

Step
→ How do I divide the job into meaningful phases?

ItemReader
→ How do I read one item at a time?

ItemProcessor
→ Where do I transform/validate an item?

ItemWriter
→ Where do I persist/output processed items?

chunk(...)
→ How do I group processing into transaction boundaries?

JobRepository
→ Where does Spring Batch keep execution state?

JobLauncher
→ How do I start a job?

JobParameters
→ What identifies a particular run/input?

JobExecution
→ What happened during an execution attempt?
```

- Spring Batch structures, executes, tracks, and can restart large batch operations — a genuinely different concern from `@Scheduled`'s "when."
- A `Job` is built from one or more `Step`s; a chunk-oriented `Step` drives an `ItemReader` → `ItemProcessor` → `ItemWriter` pipeline in groups (a "chunk"), each chunk committing as one transaction.
- A `Job` plus its `JobParameters` identifies a `JobInstance`; each attempt at running that instance is a `JobExecution`, tracked (along with each `StepExecution`) in the `JobRepository`.
- Fault tolerance (`skip`/`retry`) is opt-in via `.faultTolerant()`, and distinguishes genuinely bad items (skip) from possibly-temporary failures (retry).
- `@Scheduled` and Spring Batch commonly work together: the scheduler decides when to launch a job, Spring Batch decides how that job actually runs.

Cheat Sheet

```
// Job + chunk-oriented Step
@Bean
Job orderImportJob(JobRepository repo, Step importStep) {
    return new JobBuilder("orderImportJob", repo).start(importStep).build();
}

@Bean
Step importStep(JobRepository repo, PlatformTransactionManager tx,
                ItemReader<Order> reader, ItemProcessor<Order, Order> processor,
                ItemWriter<Order> writer) {
    return new StepBuilder("importStep", repo)
            .<Order, Order>chunk(100, tx)
            .reader(reader).processor(processor).writer(writer)
            .build();
}

// ItemProcessor: return null to filter, throw to fail/skip
ItemProcessor<Order, Order> processor() {
    return order -> order.amount().signum() < 0 ? null : order;
}

// Fault tolerance
.faultTolerant()
.skip(InvalidOrderException.class).skipLimit(10)
.retry(TransientDataAccessException.class).retryLimit(3)

// JobParameters identify a JobInstance
JobParameters params = new JobParametersBuilder()
        .addString("file", "orders.csv")
        .addLocalDate("businessDate", LocalDate.now())
        .toJobParameters();

// @Scheduled launches the job; Spring Batch runs it
@Scheduled(cron = "0 0 2 * * *")
void launch() throws Exception {
    jobLauncher.run(orderImportJob, params);
}
```

Glossary

- **Job**: the overall, reusable definition of a batch process, made of one or more Steps.
- **Step**: one phase of a Job, commonly chunk-oriented (read, process, write).
- **Chunk**: a group of items processed and committed together as one transaction.
- **ItemReader / ItemProcessor / ItemWriter**: the three roles a chunk-oriented step drives — read one item, optionally transform/validate it, write a whole chunk at once.
- **JobParameters**: the identifying input (like a file name or a business date) that, combined with a Job, defines a JobInstance.
- **JobInstance**: a specific Job run identified by its JobParameters — the same JobInstance can be attempted more than once.
- **JobExecution / StepExecution**: one attempt at running a JobInstance / a Step within it, tracked with a status (e.g. FAILED, COMPLETED).

- **JobRepository**: where Spring Batch persists execution state — job/step status, metadata, and restart-related information.
- **JobLauncher**: the component that actually starts a Job running with a given set of JobParameters.
- **Skip / Retry**: fault-tolerance options — skip tolerates a genuinely bad item (up to a limit); retry tolerates a possibly-temporary failure of an otherwise-good operation.