Mastering Spring Batch: A Comprehensive Guide

by *Reigns*

Table of Contents

1. Introduction to Spring Batch

Spring Batch is a lightweight, comprehensive framework designed for processing large volumes of data in batch jobs. It is built on top of the Spring Framework and provides necessary features for job processing, such as transaction management, retry, skip, parallel processing, and more.

2. Basic Concepts in Spring Batch

Key components of Spring Batch:
- **Job**: The entire batch process, typically made of multiple steps.
- **Step**: An individual phase within a job, like reading data, processing data, and writing data.
- **ItemReader**: Reads input data.
- **ItemProcessor**: Processes data.
- **ItemWriter**: Writes processed data.
- **JobLauncher**: Starts a job execution.
- **JobRepository**: Stores job execution details.

3. Batch Job Configuration in Spring Batch

In Spring Batch, jobs are configured with specific beans for each component like the reader, processor, and writer. Here's how to configure them:

```
@Configuration
@RequiredArgsConstructor
public class batchConfig {

    private final JobRepository jobRepository;
    private final StudentRepository repository;
    private final PlatformTransactionManager
platformTransactionManager;

    @Bean
    public FlatFileItemReader<Student> itemReader(){

        FlatFileItemReader<Student> itemReader = new
FlatFileItemReader<>();
        itemReader.setResource(new
FileSystemResource("src/main/resources/students.csv"));
        itemReader.setName("csvReader");
        itemReader.setLinesToSkip(1);
        itemReader.setLineMapper(lineMapper());
        return itemReader;
    }

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

    @Bean
    public RepositoryItemWriter<Student> writer(){
        RepositoryItemWriter<Student> writer = new
RepositoryItemWriter<>();
        writer.setRepository(repository);
        writer.setMethodName("save");

        return writer;
    }

    @Bean
    public Step importStep(){
        return new StepBuilder("csvImport", jobRepository)
                .<Student, Student>chunk(1000,
platformTransactionManager)
                .reader(itemReader())
                .processor(processor())
                .writer(writer())
                .build();
    }

    @Bean
    public Job runJob(){
```

```
        return new JobBuilder("importstudents", jobRepository)
                .start(importStep())
//                .next()  if you have more than one steo
                .build();
    }

    private LineMapper<Student> lineMapper(){
        DefaultLineMapper<Student> lineMapper = new
DefaultLineMapper<>();
        DelimitedLineTokenizer lineTokenizer = new
DelimitedLineTokenizer();
        lineTokenizer.setDelimiter(",");
        lineTokenizer.setStrict(false);
        lineTokenizer.setNames("id","firstname","lastname", "age");

        BeanWrapperFieldSetMapper<Student> fieldSetMapper = new
BeanWrapperFieldSetMapper<>();
        fieldSetMapper.setTargetType(Student.class);
        lineMapper.setLineTokenizer(lineTokenizer);
        lineMapper.setFieldSetMapper(fieldSetMapper);

        return lineMapper;

    }
}
```

Explanation:

The code above is a user defined config.java class.

- The @Configuration annotation tell springboot that we will have @Bean here. Classes marked with @Configuration are automatically loaded by springboot.
- @RequiredArgConstructor is a Lombok annotation that creates a construction with the required arguments. A springboot alternative would be to use @Autowired in the properties.
- Private final JobRepostory is an interface that is used to monitor the batch activity (spring batch creates it's own table when it runs). It is responsible for **storing and tracking** batch job execution details.
- PlatformTransactionManager manages the transaction process in the batch

- ItemReader() in this case returns a FlatFileItemReader of type Student. Flat files are files that do not have any relationship. They include CSV, Logs etc. this method sets the source of the file which in this case is a file stored in the resource folder (another alternative would be to call the ClasspathResource), give the reader a name , skips the first line which is usually the title and sets a Linemapper.

- Processor(): this is where the processing of each read data happens. You can for instance transform each data like student.firstname.uppercase() here.

- writer() : this method writes the processed data to any output stream such as a database, a file, etc. In the above example, we used a RepositoryItemWriter class that implements the ItemWriter interface (which is a functional interface) that saves the processed data to the student table. Other ItemWriters aside the RepositoryItemWriter (CTRL+ALT+F7) include: RedisitemWriter, MongoItemWriter, ListitemWriter, SimpleMailMessageItemWriter, JpaItemWriter etc

- importStep():Remember that we mentioned that a batch process s a Job and is made up of steps. This method is where the actual batch activity takes place. Here we divide the input data into chunkSize of 1000, then we read, process, write and then run the build()

- runJob(): when the user calls a jobLauncher, the runJob method is called which starts with the inportStep and then any other step specified in the next() method. We can also listen to events as they are performed in this method.

5. Exception Handling: Retry & Skip Logic

In Spring Batch, retry and skip logic are used for handling errors in a robust way. You can configure how to retry operations that fail temporarily (like database timeouts) or skip operations that fail due to permanent issues (like bad data). For example:

```
@Bean
public Step stepWithRetryAndSkip() {
   return new StepBuilder("stepWithRetryAndSkip", jobRepository)
      .<Student, Student>chunk(10, platformTransactionManager)
      .reader(itemReader())
      .processor(itemProcessor())
      .writer(itemWriter())
      .faultTolerant()
      .retry(SQLException.class)   // Retry if SQL error occurs
      .retryLimit(3)            // Retry up to 3 times
      .skip(NumberFormatException.class)  // Skip invalid data format
      .skipLimit(5)             // Allow skipping up to 5 records
      .build();
}
```

In case of an exception, you can **retry** the operation multiple times or **skip** invalid records and continue processing.

6. Handling `ConstraintViolationException` in `ItemWriter`

When saving multiple records using a method like `saveAll()` in Spring Data repositories, you might encounter a situation where some records violate constraints (e.g., a duplicate key or a missing required field).Spring Batch allows you to handle such situations gracefully.

Handling Exception in `ItemWriter` (SaveMany with Constraint Violation)

If 3 out of 10 records cause a constraint violation (like a duplicate), Spring Batch should **continue saving** the remaining records while logging the failed ones. You can configure this logic as follows:

```
@Bean
public RepositoryItemWriter<Student> writer() {
  RepositoryItemWriter<Student> writer = new RepositoryItemWriter<>();
  writer.setRepository(studentRepository);
  writer.setMethodName("saveAll");
  writer.setItemCountLimit(1000);
  writer.setExceptionHandler(new ItemWriterExceptionHandler() {
    @Override
    public void handleException(Exception e, List<? extends Student> items) {
      if (e instanceof ConstraintViolationException) {
        // Log failed records, handle the exception, and continue
        logger.error("Constraint violation error: " + e.getMessage());
      } else {
        // Handle other exceptions appropriately
        logger.error("Error while saving items: " + e.getMessage());
      }
    }
  });
  return writer;
}
```

This `ItemWriter` will **attempt to save all records**, and if a `ConstraintViolationException` occurs for some records, it will **log the error** and continue with the remaining records.

In case of a permanent violation (e.g., a unique constraint violation), you can also consider skipping the record and continuing with the valid ones by adding skip logic to the step

configuration.

Will the 7 valid records save?
Yes, in this scenario, the 7 valid records will still be saved. The exception handling mechanism ensures that the batch job continues and doesn't stop due to individual failures.

To track which records were successfully saved and which ones failed due to exceptions (e.g., `ConstraintViolationException` or other errors), you need to implement custom logging or tracking mechanisms within your `ItemWriter`. Here's how you can achieve this:

1. Custom Exception Handling in the `ItemWriter`

You can add exception handling in the `ItemWriter` to **log** or **save** the failed records separately. You can maintain a list of successful and failed records, and once the step completes, you can examine which records failed.

2. Using an `ItemWriter` with Custom Logic

Modify your `ItemWriter` to handle exceptions and track the saved/failed records

```
@Bean
public RepositoryItemWriter<Student> writer() {
   RepositoryItemWriter<Student> writer = new RepositoryItemWriter<>();
   writer.setRepository(studentRepository);
   writer.setMethodName("saveAll");

   // Custom exception handling
   writer.setItemCountLimit(1000); // Optional - for limiting batch size
   writer.setExceptionHandler(new ItemWriterExceptionHandler() {
      @Override
      public void handleException(Exception e, List<? extends Student> items) {
         if (e instanceof ConstraintViolationException) {
            // Log failed records
            logger.error("Constraint violation error: " + e.getMessage());
            logFailedRecords(items);
         } else {
            // Log other exceptions
            logger.error("Error while saving items: " + e.getMessage());
         }
      }
   });

   return writer;
}

private void logFailedRecords(List<? extends Student> items) {
   for (Student student : items) {
```

```
        // Log failed records with custom error message
        logger.error("Failed to save record: " + student);
    }
}
```

3. Tracking Success and Failure

You can also create a **custom log** that stores the status of each record. For instance, after processing each chunk, you could log the successful records and those that failed.

You can achieve this by overriding the `write()` method and implementing a custom `ItemWriter`. You will manually keep track of each record's status.

```
public class CustomItemWriter implements ItemWriter<Student> {

    private final List<Student> successfulRecords = new ArrayList<>();
    private final List<Student> failedRecords = new ArrayList<>();

    @Override
    public void write(List<? extends Student> items) throws Exception {
        for (Student student : items) {
            try {
                // Attempt to save the student record
                studentRepository.save(student);
                successfulRecords.add(student);
            } catch (ConstraintViolationException e) {
                // Log failure for the record
                failedRecords.add(student);
                logger.error("Failed to save student due to constraint violation: " + student);
            } catch (Exception e) {
                // Catch other types of exceptions
                failedRecords.add(student);
                logger.error("Failed to save student due to error: " + student);
            }
        }

        // Optionally, you can process the lists of successful and failed records here,
        // or after the chunk is processed.
        logResults();
    }

    private void logResults() {
        logger.info("Successfully saved " + successfulRecords.size() + " records.");
        logger.info("Failed to save " + failedRecords.size() + " records.");
        // You can log the failed records or save them to a separate file/database if needed.
    }
```

```
    public List<Student> getSuccessfulRecords() {
        return successfulRecords;
    }

    public List<Student> getFailedRecords() {
        return failedRecords;
    }
}
```

4. Step-Level Logging

You can also implement logging at the **step level** (using listeners) to track the chunk processing, successful and failed records.

Example Using `ItemWriteListener`:

```
@Bean
public Step processStep() {
    return stepBuilderFactory.get("processStep")
            .<Student, Student>chunk(100)
            .reader(itemReader())
            .processor(itemProcessor())
            .writer(itemWriter())
            .listener(new ItemWriteListener<Student>() {
                @Override
                public void beforeWrite(List<? extends Student> items) {
                    // Optionally, log or inspect records before they are written
                }

                @Override
                public void afterWrite(List<? extends Student> items) {
                    // Log successfully written records
                    logger.info("Successfully saved " + items.size() + " records.");
                }

                @Override
                public void onWriteError(Exception exception, List<? extends Student> items) {
                    // Log failed records
                    logger.error("Error occurred while writing records: " + exception.getMessage());
                    items.forEach(item -> logger.error("Failed to save record: " + item));
                }
            })
            .build();
}
```

5. Retry Logic and Skipping Records

In case of an exception that you want to **retry** or **skip**, you can add **retry** and **skip** logic to ensure that only the invalid records are skipped and the valid ones continue to process.

For instance, skipping a record after a certain number of failures:

```
@Bean
public Step processStepWithSkip() {
   return stepBuilderFactory.get("processStepWithSkip")
         .<Student, Student>chunk(100)
         .reader(itemReader())
         .processor(itemProcessor())
         .writer(itemWriter())
         .faultTolerant()
         .skip(ConstraintViolationException.class)  // Skip constraint violation errors
         .skipLimit(5)  // Skip up to 5 records
         .build();
}
```

6. Storing Failed Records to a Separate File or Database

If you want to **persist failed records** (instead of just logging them), you can write them to a separate file or database:

```
@Bean
public ItemWriter<Student> failedRecordsWriter() {
   return new ItemWriter<Student>() {
      @Override
      public void write(List<? extends Student> items) throws Exception {
         // Write the failed records to a separate file or database
         failedRecordRepository.saveAll(items);
      }
   };
}
```

Parallel Processing & Multi-Threaded Steps in Spring Batch

Spring Batch provides **parallel processing** to increase performance by processing multiple chunks at the same time. There are multiple ways to achieve tis such as multi-threaded steps (using taskExecutor), partitioning (splitting work across multiple threads), remote chunking (processing on distributed nodes) and remote partitioning (Distributing workload across multiple servers).

Multi-threading step: with taskExecutor () is used to run parallel threads in the step. For instance:

```
Public Step multithreadStep(){
  Return new StepBuilder(“nameOFStep”, jobRepository)
        .<Student, Student>chunk(10, platformTransactionManager)
        .reader(reader())
        .processor(processor())
        .writer(writer())
        .taskExecutor( new SimpleAsyncTaskExecutor())
        .build()
        .
}
```

In the above example, the step runs a chunk of 10 data and the taskExecutor spins up a new thread for each chunk. Another was would be to use an ThreadPoolTaskExecutor class.

Partitioning (Recommended for Scaling)

- Breaks a large dataset into smaller **partitions** and processes them in parallel.
- Each partition runs as a separate **Step execution**.

- **Example: Using a Partitioner**

```
@Bean
public Step partitionedStep() {
   return new StepBuilder("partitionedStep", jobRepository)
        .partitioner("workerStep", myPartitioner())
        .step(workerStep())
        .gridSize(5) // Runs 5 parallel partitions
        .taskExecutor(new SimpleAsyncTaskExecutor())
        .build();
}
```

When to use?

- When a dataset is **too large** to process in a single step.
- Distributing workload across multiple threads or even **remote servers**.