Spring Batch

- Spring Batch is a lightweight, open-source framework used for batch processing in Java applications.
- It is part of the larger Spring ecosystem and focuses on the development of robust, scalable, and high-performance batch processing applications.
- key aspects of Spring Batch :

Batch Job Execution: Spring Batch allows you to define jobs consisting of one or more steps. Each step can perform a specific task such as reading data, processing it, and writing the output.

Chunk-Oriented Processing: The framework supports reading data in chunks, processing the chunks, and then writing the processed data in batches, which is efficient for large datasets.

Item Readers and Writers: Spring Batch provides built-in ItemReader and ItemWriter implementations for reading and writing data from various sources like databases, files, or other systems.

Transaction Management: Spring Batch integrates seamlessly with Spring's transaction management system. It ensures consistency and can handle retries and rollbacks in case of failures.

Job and Step Configuration: Jobs and steps are configured using XML or Java-based configuration. The jobs can have flow control (e.g., decision-based flows, skip steps, and retry mechanisms).

Parallel Processing: It supports parallel processing, allowing tasks to be executed concurrently across multiple threads or machines, which improves performance.

Fault Tolerance: It provides mechanisms for handling failures such as retry, skip, and recovery for both individual items and entire steps in the batch process.

Typical Use Cases:

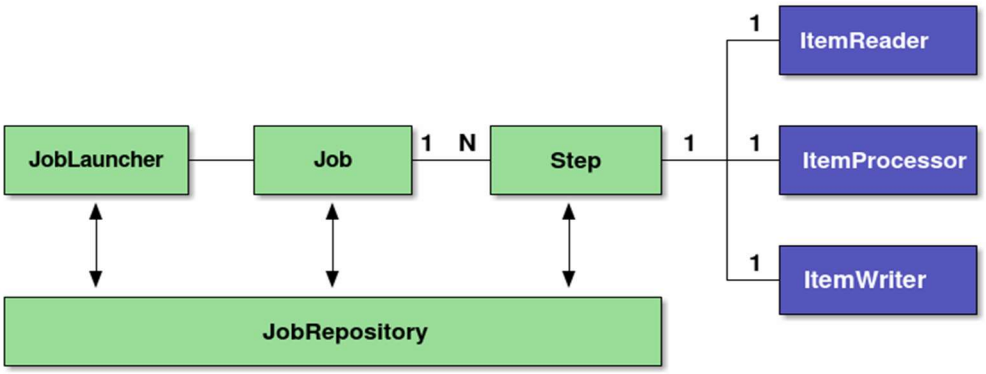
Data Migration: Moving data between systems, for example, from one database to another.

ETL Jobs: Extracting data from sources (files, databases), transforming it, and loading it into another system.

Scheduled Reports: Generating large reports on a scheduled basis.

Processing Large Data Volumes: Efficiently processing and writing large datasets in chunks.

❖ Spring Batch Architecture :



Job : A single execution unit that contains a series of processes for batch application in Spring Batch.

Step: A unit of processing which constitutes Job. 1 job can contain 1~N steps Reusing a process, parallelization, conditional branching can be performed by dividing 1 job process in multiple steps. Step is implemented by either chunk model or tasklet model.

JobLauncher : An interface for running a Job. JobLauncher can be directly used by the user, however, a batch process can be started simply by starting `CommandLineJobRunner` from java command. `CommandLineJobRunner` undertakes various processes for startingJobLauncher.

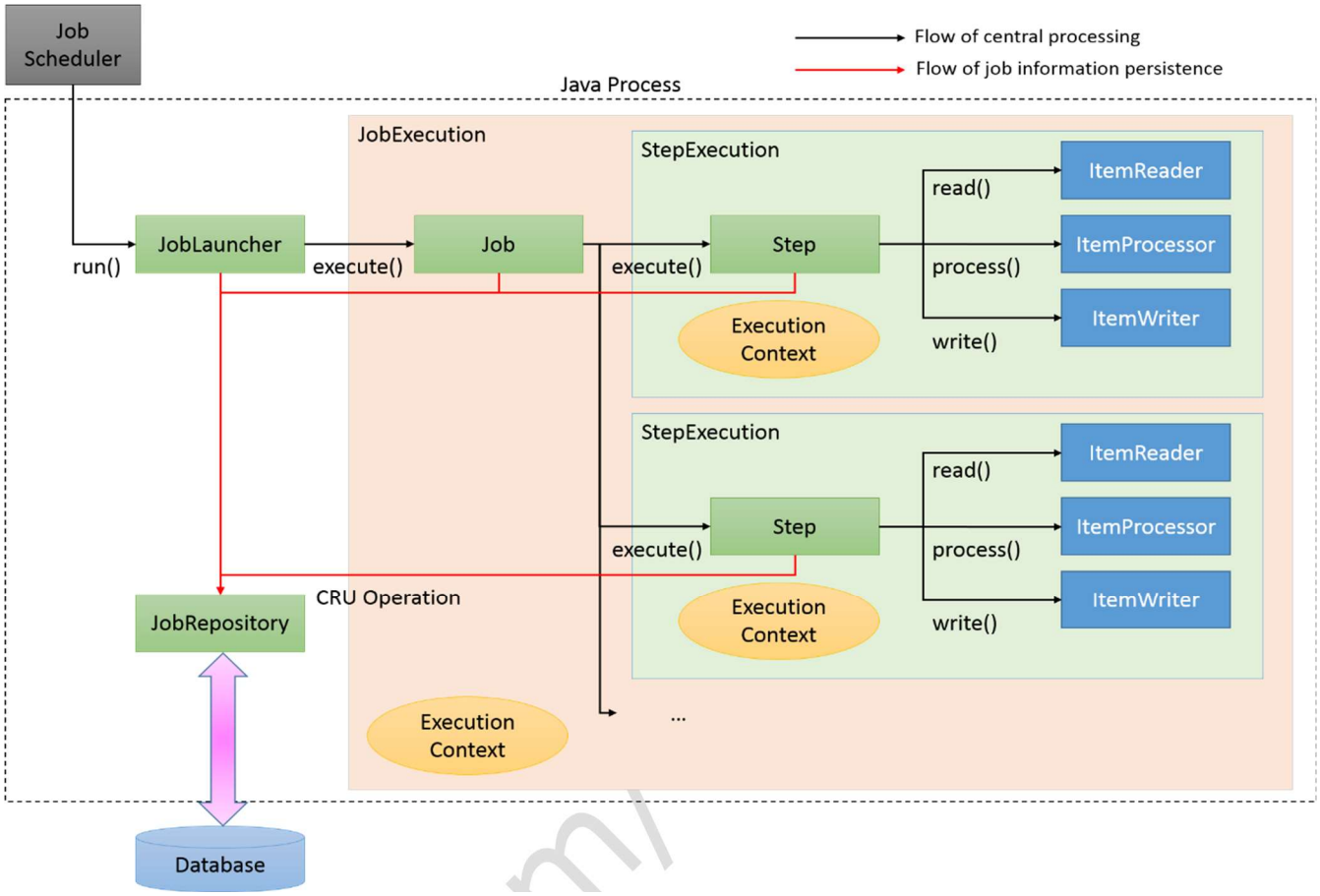
ItemReader & ItemProcessor & ItemWriter :

- **ItemReader** and **ItemWriter** responsible for data input and output are often the processes that perform conversion of database and files to Java objects and vice versa, a standard implementation is provided by Spring Batch.
- In batch applications which perform input and output of data from file and database, conditions can be satisfied just by using standard implementation of Spring Batch as it is.
- ItemProcessor which is responsible for processing data implements input check and business logic.

JobRepository :

A system to manage condition of Job and Step. The management information is persisted on the database based on the table schema specified by Spring Batch.

Overall Process Flow :



Main processing flow

1. JobLauncher is initiated from the job scheduler.
2. Job is executed from JobLauncher.
3. Step is executed from Job.
4. Step fetches input data by using ItemReader.
5. Step processes input data by using ItemProcessor.

A flow for persisting job information

1. JobLauncher registers JobInstance in Database through JobRepository.
2. JobLauncher registers that Job execution has started in Database through JobRepository.
3. JobStep updates miscellaneous information like counts of I/O records and status in Database through JobRepository.
4. JobLauncher registers that Job execution has completed in Database through JobRepository.

Important Tips for Batch processing :

- When working with Spring Batch, every developer needs to know these point for performance tuning.
  - Adjusting the chunk size
  - Adjusting the Fetch Size (Buffer Size)
  - Handling File Operation in effective manner.
  - Parallel processing, Multiple Processing **
  - Distributed Processing (Executing Jobs across multiple machines) **

Spring Scheduling

==========

Spring Scheduling in Java refers to the functionality provided by the Spring Framework to allow the execution of tasks at specific times or intervals, without needing a separate scheduling framework. This can be especially useful for background tasks, such as sending periodic emails, cleanup operations, or database maintenance.

1. Using @Scheduled Annotation

The most popular way to define scheduled tasks in Spring is by using the @Scheduled annotation. This annotation allows methods to be executed periodically or at fixed intervals.

Key Features:

- Fixed Rate: The method is executed at a fixed interval, measured from the start time of the previous execution.
- Fixed Delay: The method is executed after the previous execution finishes, with a specified delay between executions.
- Cron Expression: Use a cron expression to schedule tasks at specific times.

Example of @Scheduled Usage

Here is an example of how to use the @Scheduled annotation in Spring.

```
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;
@Component
public class ScheduledTasks {
    // Executes every 5 seconds
    @Scheduled(fixedRate = 5000)
    public void fixedRateTask() {
        System.out.println("Task executed with fixed rate every 5 seconds");
    }

    // Executes 5 seconds after the previous task ends
    @Scheduled(fixedDelay = 5000)
    public void fixedDelayTask() {
        System.out.println("Task executed with fixed delay of 5 seconds");
    }

    // Executes at a specific time using a cron expression
    @Scheduled(cron = "0 0 * * * ?") // Executes every hour on the hour
    public void cronTask() {
        System.out.println("Task executed every hour on the hour");
    }
}
```

2. Enabling Scheduling in Spring

For scheduling to work, you need to enable it in your Spring configuration class by using the @EnableScheduling annotation.

```
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.annotation.EnableScheduling;

@Configuration
@EnableScheduling
public class SchedulingConfig {
}
```

3. Cron Expressions

A cron expression allows you to specify more complex scheduling. It consists of five or six fields separated by spaces. The fields represent seconds (optional), minutes, hours, day of the month, month, and day of the week.

Example Cron Expression Format:

```
* * * * * *
| | | | | |
| | | | | └─ Day of the week (0 - 6) (Sunday=0)
| | | | └──── Month (1 - 12)
| | | └────── Day of the month (1 - 31)
| | └──────── Hour (0 - 23)
| └────────── Minute (0 - 59)
└──────────── Second (0 - 59)
```

Example:

```
@Scheduled(cron = "0 0 12 * * ?") // Executes every day at 12 PM
public void executeEveryDayAtNoon() {
    System.out.println("Scheduled task running at 12 PM every day.");
}
```

4. TaskExecutor with Scheduling

If you want to run tasks concurrently, you can define a TaskExecutor and schedule tasks to be executed in different threads. This helps improve performance if tasks are long-running or I/O intensive.

```
import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class SchedulingConfig {

    @Bean
    public ThreadPoolTaskExecutor taskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(5); // The number of core threads
        executor.setMaxPoolSize(10); // Maximum number of threads
        executor.setQueueCapacity(25); // The capacity of the task queue
        executor.setThreadNamePrefix("scheduled-task-");
        executor.initialize();
        return executor;
    }
}
```

5. @Async with Scheduling

You can combine @Async and @Scheduled to run a task asynchronously.

```java
import org.springframework.scheduling.annotation.Async;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

@Component
public class AsyncScheduledTasks {

    @Async
    @Scheduled(fixedRate = 5000)
    public void asyncTask() {
        // Simulating long-running task
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Async task executed");
    }
}
```

6. Handling Task Exceptions

If a task fails, you can handle exceptions within your scheduled task by using a try-catch block or by using an ErrorHandler if needed.

7. Disabling Scheduling

You can stop a scheduled task by removing the @Scheduled annotation or by managing the task manually using TaskScheduler beans or scheduling frameworks such as Quartz.

8. Using TaskScheduler for More Control

For advanced scheduling scenarios, you can use TaskScheduler instead of @Scheduled. This gives more flexibility for creating, scheduling, and canceling tasks programmatically.

```java
import org.springframework.scheduling.TaskScheduler;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class CustomScheduler {

    @Autowired
    private TaskScheduler taskScheduler;

    public void scheduleTask() {
        taskScheduler.schedule(() -> System.out.println("Task executed!"), new Date());
    }
}
```