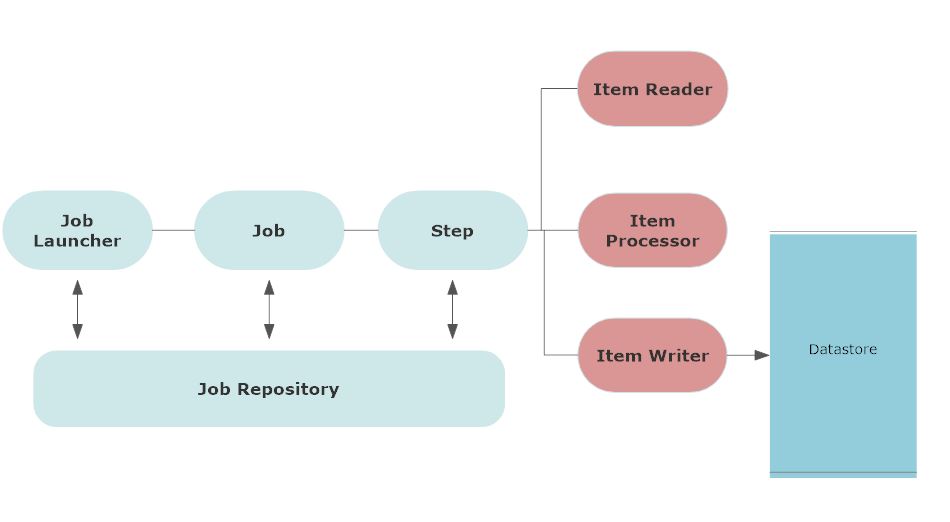
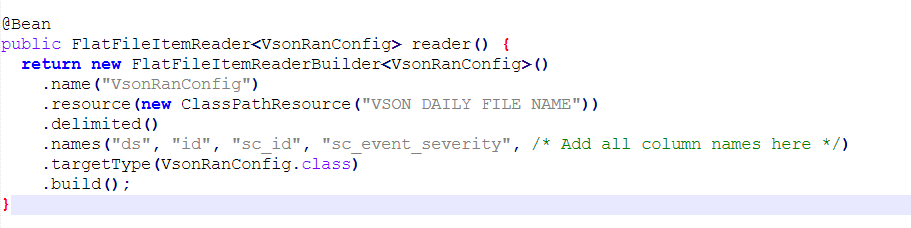
**Spring Batch**

Spring Batch is a lightweight, comprehensive framework for building batch processing applications in Java. It provides robust features for processing large volumes of data efficiently, handling failures gracefully, and managing complex processing workflows. In this document, we will explore how Spring Batch is implemented, focusing on various components such as **ItemReader, ItemProcessor, ItemWriter, Job, Step, JobRepository, and JobLauncher.**

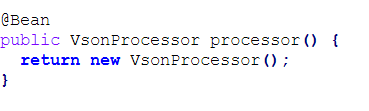
**Spring Batch Architecture**  
  
**Item Reader**

The ItemReader interface in Spring Batch is responsible for reading data from input sources such as files, databases, or streams. It reads data in chunks and passes it to the ItemProcessor for processing. The significance of the ItemReader lies in its ability to efficiently fetch data in a scalable manner, enabling batch processing of large datasets.



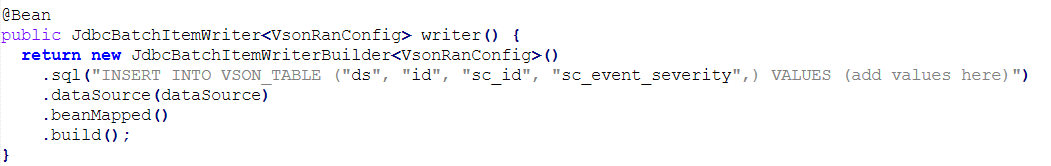
**Item Processor**

The ItemProcessor interface is responsible for processing each item read by the ItemReader. It allows for transformation or manipulation of data before passing it to the ItemWriter for writing to the output destination. The ItemProcessor is crucial for implementing business logic, data validation, or enrichment during batch processing.



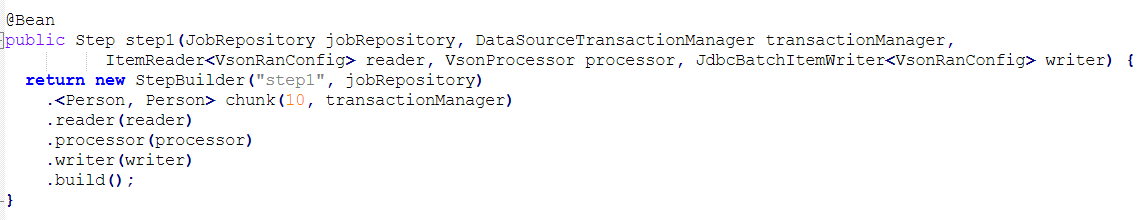
**Item Writer**

The ItemWriter interface is responsible for writing processed data to the output destination, such as databases, files, or message queues. It receives input data from the ItemProcessor and writes it in chunks to improve performance and reduce database load. The ItemWriter ensures data persistence and consistency during batch processing.



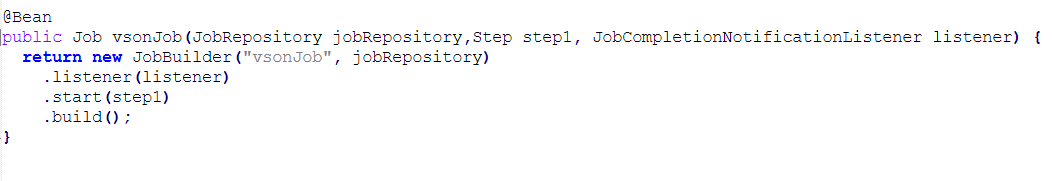
**Step**

A Step in Spring Batch represents a single atomic unit of work within a job. It consists of an ItemReader, ItemProcessor, and ItemWriter configured to process a chunk of data. Steps are executed sequentially within a job, and their execution can be controlled and monitored independently.



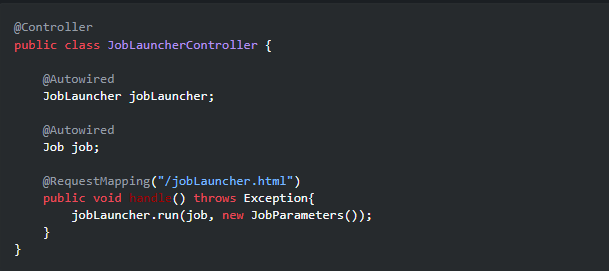
**Job**

A Job in Spring Batch represents a logical unit of work, which consists of one or more steps. It orchestrates the execution of steps and provides mechanisms for controlling job execution, such as starting, stopping, and monitoring.



**JobLauncher**

The JobLauncher interface is responsible for launching job executions in Spring Batch. It provides methods for starting and executing jobs programmatically. The JobLauncher interacts with the JobRepository to manage job executions and handle job lifecycle events.



**Extra Tables Created by Spring Batch:**

1.**BATCH\_JOB\_INSTANCE**: This table stores information about each unique job instance launched. It includes fields such as job name and job parameters.

2.**BATCH\_JOB\_EXECUTION**: This table contains information about each job execution, including start and end times, status, and exit codes.

3.**BATCH\_JOB\_EXECUTION\_PARAMS**: This table stores the parameters used for each job execution. It provides a mapping between job executions and their parameters.

4.**BATCH\_STEP\_EXECUTION**: This table stores information about each step execution within a job. It includes start and end times, status, commit counts, and read/write counts.

5.**BATCH\_STEP\_EXECUTION\_CONTEXT**: This table stores the execution context data for each step execution. It includes any user-defined data that needs to be persisted between steps.

6.**BATCH\_JOB\_SEQ**: This sequence table is used to generate unique identifiers for job instances.

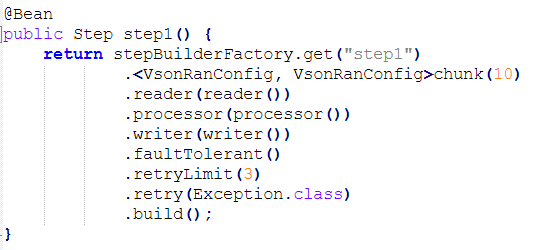
7.**BATCH\_JOB\_EXECUTION\_SEQ:** This sequence table is used to generate unique identifiers for job executions.

8.**BATCH\_STEP\_EXECUTION\_SEQ**: This sequence table is used to generate unique identifiers for step executions.

**Handling Failure Scenarios :**

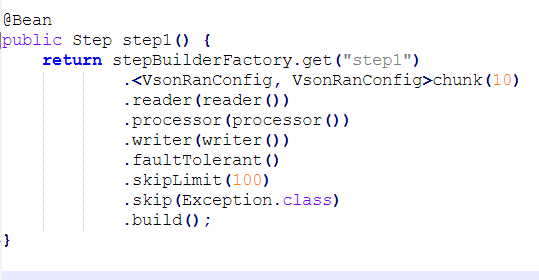
**Retry Logic**

Spring Batch allows you to configure retry logic for failed steps. You can specify the number of retry attempts and conditions for retries. If a step fails, Spring Batch will automatically retry it according to the configured retry policy.



**Skip Logic**

Spring Batch provides skip logic to handle items that cannot be processed due to errors. You can configure skip policies to skip certain items or exceptions. Failed items are skipped, and processing continues with the next item.



Here, skip Limit(100) indicates that the step step1 will allow up to 100 exceptions to be skipped before it stops skipping further exceptions and fails the job.

If more than 100 exceptions occur during the execution of this step, the job will fail, and further processing will not continue.