Spring Batch

Use Case :

Data Integration and ETL (Extract, Transform, Load): Spring Batch can be used to extract data from various sources, such as databases, flat files, or web services. It allows developers to apply transformations and business rules to the data before loading it into a target system. For example, you can use Spring Batch to extract customer data from a CSV file, perform data cleansing and validation, and load it into a database.

Report Generation: Spring Batch can be used to generate reports by processing large amounts of data. You can define jobs that fetch data from databases or other sources, aggregate and analyze it, and generate reports in various formats, such as PDF, Excel, or HTML. For instance, you can create a daily sales report by processing transactional data and summarizing it in a structured format.

Batch Processing of Financial Transactions: Many financial applications require batch processing to handle large volumes of financial transactions efficiently. Spring Batch can be used to process transactions in batches, apply business rules and validations, and update account balances or generate financial reports. It provides features like transaction management, retry mechanisms, and error handling to ensure data integrity and reliability.

Data Migration and Conversion: When migrating data from one system to another, Spring Batch can be used to handle the data conversion and migration tasks. It allows you to read data from the source system, transform it according to the target system's requirements, and load it into the new system. For example, you can use Spring Batch to migrate data from a legacy system to a modern database.

Batch Processing in Enterprise Applications: Many enterprise applications have background processes or scheduled jobs that perform batch processing tasks, such as sending notifications, generating invoices, or updating caches. Spring Batch provides a robust and scalable framework to handle these batch processing requirements. It offers features like job scheduling, parallel processing, and restartability to ensure reliable and efficient batch processing.

Features of Spring Batch:

Spring Batch offers a wide range of features that make it a powerful framework for building batch processing applications. Here are some key features of Spring Batch:

1. Job Processing: Spring Batch provides a structured and organized way to define and process batch jobs. A job consists of multiple steps, each representing a specific task or processing stage. Jobs can be executed sequentially or in parallel, and Spring Batch manages the execution and coordination of these steps.

2. Step Processing: Steps are the building blocks of a job in Spring Batch. Each step represents a specific task, such as reading data, processing it, and writing the results. Spring

Batch provides a variety of pre-built step implementations for common tasks like reading from databases, files, or web services, and writing to various destinations.

3. Chunk-Based Processing: Spring Batch supports chunk-based processing, where large amounts of data are read, processed, and written in chunks. This approach helps optimize performance and memory usage, as it allows processing large datasets efficiently in smaller, manageable chunks.

4. Item Readers and Writers: Spring Batch provides a range of item readers and writers that handle reading input data and writing output data. It includes readers for databases (JDBC), files (FlatFileItemReader), XML (StaxEventItemReader), and more. Similarly, there are writers for databases, files, and XML. Additionally, custom item readers and writers can be developed to handle specific data sources and destinations.

5. Item Processors: Item processors allow you to apply business logic and transformations to the data being processed. They can perform validation, filtering, enrichment, or any other required processing on individual items. Item processors can be chained together, allowing for complex data manipulations.

6. Error Handling and Retry: Spring Batch provides robust error handling capabilities. It allows you to configure error handling strategies, such as skipping faulty records, retrying failed items, or stopping the job upon encountering errors. You can also define custom error handling logic to handle specific error scenarios.

7. Transaction Management: Spring Batch integrates with the Spring Framework's transaction management capabilities. It provides transaction support at both the job and step levels, ensuring data consistency and integrity. Transactions can be managed using various strategies, such as local or distributed transactions.

8. Restart and Recovery: Spring Batch supports job restartability and recovery, which is crucial in batch processing scenarios. It allows jobs to be restarted from a failed or interrupted state, picking up from where they left off. This feature ensures that batch jobs can recover from unexpected failures and continue processing without duplicating already processed data.

9. Job Scheduling: Spring Batch can be integrated with various job scheduling tools, such as Quartz or Spring Integration, to schedule the execution of batch jobs at specific times or intervals. This feature enables automating the execution of batch jobs as part of a larger system or enterprise workflow.

10. Monitoring and Administration: Spring Batch provides features for monitoring and administering batch jobs. It includes support for logging, metrics, and statistics about job execution. Additionally, Spring Batch integrates with management tools like Spring Boot Actuator, which offers endpoints for monitoring and managing batch jobs.

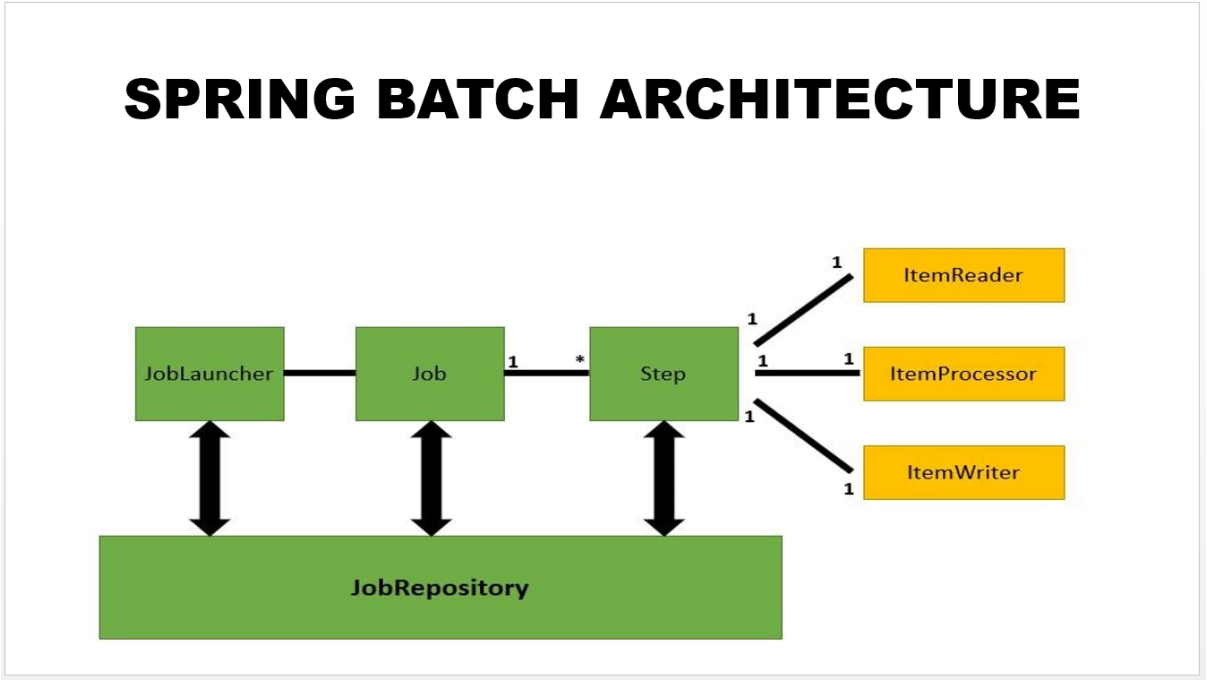
These are some of the key features provided by Spring Batch that make it a robust and efficient framework for building batch processing applications. The framework's flexibility,

scalability, and extensibility make it suitable for a wide range of batch processing requirements.

How to Start with Spring Batch?

Required dependency :

```
dependencies {
        implementation 'org.springframework.boot:spring-boot-starter-batch'
}
```



JobRepository in Spring batch takes care of all the CRUD (create, read, update, and delete) operations and ensures persistence. It does this for JobLauncher, Job, and Step. In the above code, we are creating a Job with one Step and that Step in-turn has an ItemReader, an ItemProcessor, and an ItemWriter.

A JobLauncher uses the JobRepository to create new JobExecution objects and run them. Job and Step implementations later use the same JobRepository for basic updates of the same executions during the running of a Job. The basic operations suffice for simple scenarios.

Spring Batch follows the traditional batch architecture where a job repository does the work of scheduling and interacting with the job. A job can have more than one step. And every step typically follows the sequence of reading data, processing it and writing it.

https://docs.spring.io/spring-batch/docs/current/reference/html/readersAndWriters.html

Batch Configuration File:

```
@Configuration
public class BatchConfiguration {

   @Autowired
   private JobLauncher jobLauncher;

   @Autowired
   private DataSource dataSource;

   @Bean
   public ItemReader<User> itemReader() {
      return new JdbcCursorItemReaderBuilder<User>()
            .name("userItemReader")
            .dataSource(dataSource)
            .sql("select p.first_name,p.last_name,pc.contact from user p left join user_contact
pc on p.id = pc.user_id;")
            .rowMapper(new UserMapper())
            .build();
   }

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

   @Bean
   public JdbcBatchItemWriter<User> writer(DataSource dataSource) {
      return new JdbcBatchItemWriterBuilder<User>()
            .itemSqlParameterSourceProvider(new
BeanPropertyItemSqlParameterSourceProvider<>())
            .sql("INSERT INTO user_details (first_name, last_name, contact) VALUES
(:firstName, :lastName, :contact)")
            .dataSource(dataSource)
            .build();
   }

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

```
    }

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

}
```

To auto create a batch default table in mysql

```
spring.batch.jdbc.initialize-schema=always
```