SPRING BATCH

1. Batch Processing

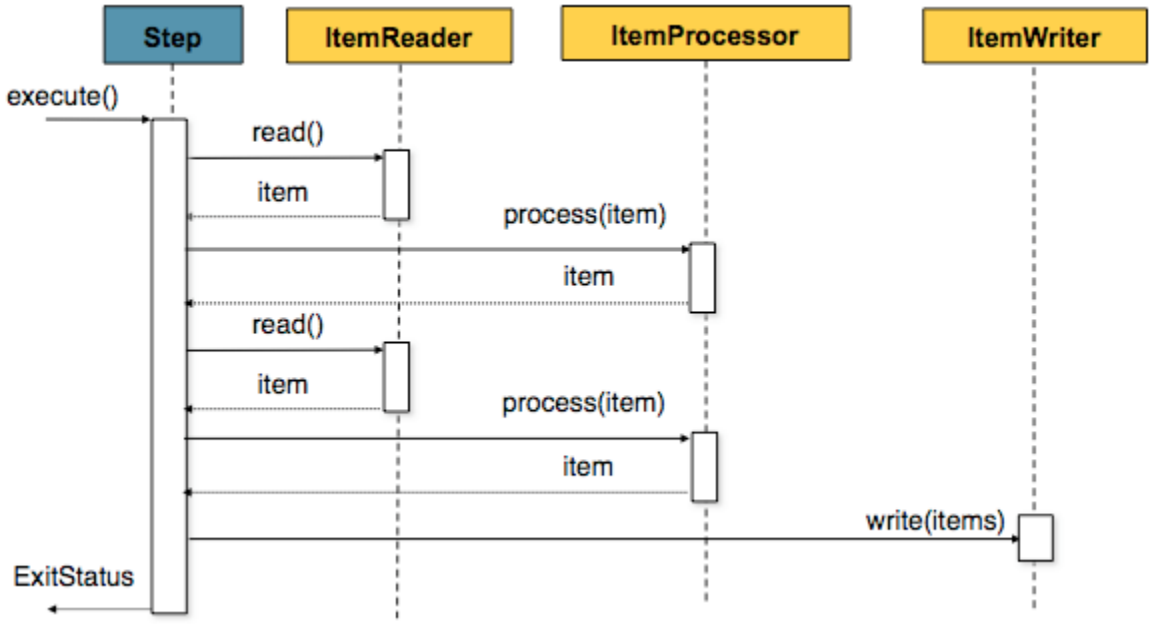
- Batch jobs are the tasks that are executed periodically or when resource usage is low, and they often process large amounts of information such as log files, database records, or images.
- Batch processing refers to running batch jobs on a computer system.
- A batch job can be completed without user intervention.
- For example, consider a telephone billing application that reads phone call records from the enterprise information systems and generates a monthly bill for each account.

1.1. 'Steps' in Batch Processing

- A step is an independent and sequential phase of a batch job. Batch jobs contain chunk-oriented steps and task-oriented steps.

1.1.1. Chunk-oriented Steps

- Chunk-oriented steps (chunk steps) process data by reading items from a data source, applying some business logic to each item, and storing the results.
- Chunk steps read and process one item at a time and group the results into a chunk.
- The results are stored when the chunk reaches a configurable size.
- Chunk-oriented processing makes storing results more efficient



1.1.2. Task-oriented Steps

- Task-oriented steps (task steps) execute tasks other than processing items from a data source. Examples include creating or removing directories, moving files, creating or dropping database tables, configuring resources, and so on. Task steps are not usually long-running compared to chunk steps.

1.2. Parallel Processing

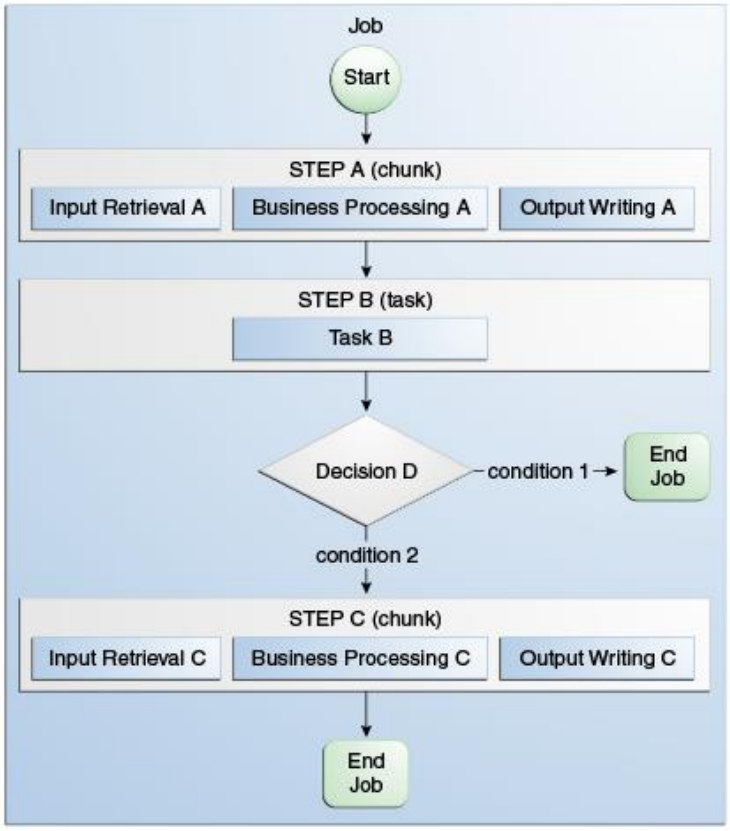
- Steps that do not depend on each other can run on different threads.
- Chunk-oriented steps where the processing of each item does not depend on the results of processing previous items can run on more than one thread.
- Batch frameworks provide mechanisms for developers to define groups of independent steps and to split chunk-oriented steps in parts that can run in parallel.

1.3. Status

- Batch frameworks keep track of a status for every step in a job.
- The status indicates if a step is running or if it has completed. If the step has completed, the status indicates one of the following:
    - The execution of the step was *successful*.
    - The step was *interrupted*.
    - An *error* occurred in the execution of the step.

1.4. Decision

- Decision elements use the exit status of the previous step to determine the next step or to terminate the batch job.
- Decision elements set the status of the batch job when terminating it.
- Like a step, a batch job can terminate successfully, be interrupted, or fail.



2. Spring Batch

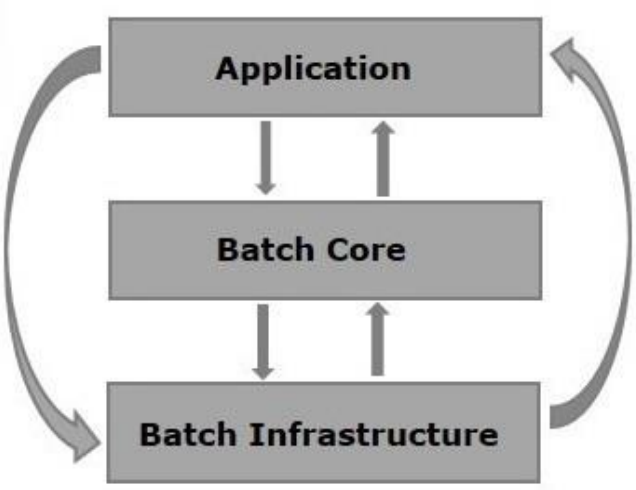
- Spring batch is a lightweight framework which is used to develop Batch Applications that are used in Enterprise Applications.

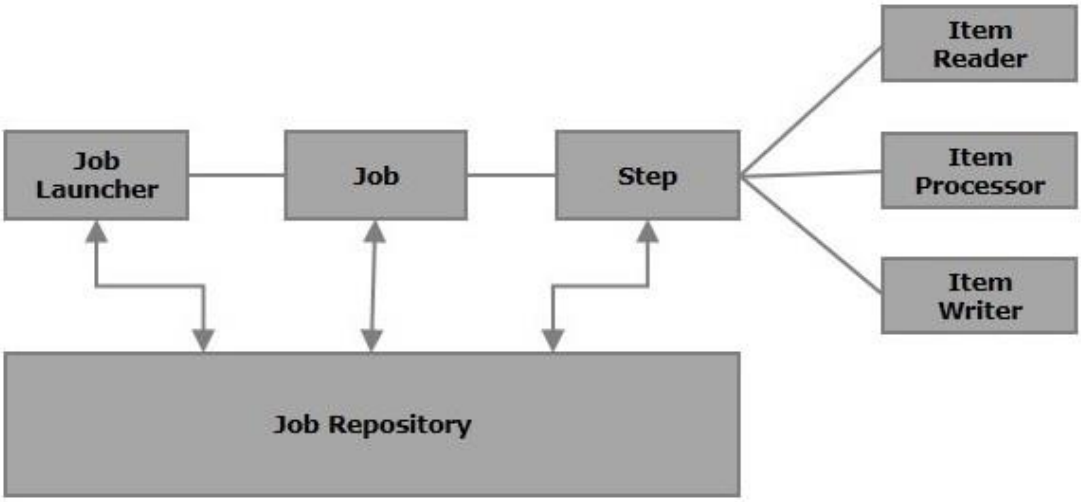
2.1 : Features

- Flexibility − Spring Batch applications are flexible. You simply need to change an XML file to alter the order of processing in an application.
- Maintainability − A Spring Batch job includes steps and each step can be decoupled, tested, and updated, without affecting the other steps.
- Scalability − Using the portioning techniques, you can scale the Spring Batch applications. These techniques allow you to −
    - Execute the steps of a job in parallel.
    - Execute a single thread in parallel.
- Reliability − In case of any failure, you can restart the job from exactly where it was stopped, by decoupling the steps.
- Support for multiple file formats.
- Multiple ways to launch a job − You can launch a Spring Batch job using web applications, Java programs, Command Line, etc.
- Automatic retry after failure.

3. Architecture of Spring Batch

- Application − This component contains all the jobs and the code we write using the Spring Batch framework.
- Batch Core − This component contains all the API classes that are needed to control and launch a Batch Job.
- Batch Infrastructure − This component contains the readers, writers, and services used by both application and Batch core components.




Item Reader
Job Launcher
Job
Step
Item Processor
Item Writer
Job Repository

3. Troubleshooting

- **Problem 1 :** While using spring-batch, jpa repository's save() method was running "SELECT" command instead of "INSERT"
  Solution : Autowiring "PlatformTransactionManager" and configuring JpaTransactionManager

  JpaCongif.java

```java
package com.demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.orm.jpa.JpaTransactionManager;

import javax.sql.DataSource;

@Configuration
public class JpaConfig {

   private final DataSource dataSource;

   @Autowired
   public JpaConfig(@Qualifier("dataSource") DataSource dataSource) {
       this.dataSource = dataSource;
   }

   @Bean
   @Primary
   public JpaTransactionManager jpaTransactionManager() {
       final JpaTransactionManager transactionManager = new
JpaTransactionManager();
       transactionManager.setDataSource(dataSource);
       return transactionManager;
   }

}
```

ChunksConfig.java

```
public class ChunksConfig {
    .
    .
    .
    @Autowired
    private PlatformTransactionManager transactionManager;

    @Bean
    public JobRepository jobRepository() throws Exception {
        MapJobRepositoryFactoryBean factory = new
MapJobRepositoryFactoryBean();
        factory.setTransactionManager(transactionManager);
        return factory.getObject();
    }
    .
    .
    .
}
```

- Full Source Code : https://github.com/vipul-kumar24/spring-batch-demo/tree/master
- Solution Reference : https://jira.spring.io/browse/BATCH-2642

4. Spring Batch with Scheduler

- In this example, the job runs after every 1 min.

```
import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import java.util.Date;

@Component
@EnableScheduling
class ScheduledTasks {
   @Autowired
   JobLauncher jobLauncher;

   @Autowired
   Job job;

   @Scheduled(cron = "0 0/1 * * * *")
   public void runJob() throws Exception {
       try {
           String dateParam = new Date().toString();
           JobParameters param = new JobParametersBuilder().addString("date",
                                                   dateParam).toJobParameters();

           System.out.println(dateParam);

           JobExecution execution = jobLauncher.run(job, param);
           System.out.println("Exit Status : " + execution.getStatus());

       } catch (Exception e) {
           e.printStackTrace();
       }

   }
}
```

Full Source Code : https://github.com/vipul-kumar24/spring-batch-demo/tree/SpringBatch-with-Scheduler

5. Spring Batch with Multithreading

- Step 1 : Make your ItemReader.read() method 'synchronized', so that only one thread have access to it at a time.

FileReader.java

```
public class FileReader implements ItemReader<File> {
...
     @Override
     public synchronized File read() throws Exception {
      ...
     }
...
}
```

- Step 2 : Add 'SimpleAsyncTaskExecutor' to Step-Builder's taskexecutor() method.

ChunkConfig.java

```
public class ChunksConfig {
...
   @Bean
   public TaskExecutor taskExecutor(){
      SimpleAsyncTaskExecutor asyncTaskExecutor=new SimpleAsyncTaskExecutor();
      asyncTaskExecutor.setConcurrencyLimit(10);//no of concurrent threads
      return asyncTaskExecutor;
   }

   @Bean
   protected Step processFiles(ItemReader<File> reader, ItemProcessor<File,
                               File> processor, ItemWriter<File> writer) {
      return steps.get("processFiles").<File, File>chunk(5000)
                  .reader(reader)
                  .processor(processor)
                  .writer(writer)
                  .taskExecutor(taskExecutor())
                  .build();
        }
...
}
```

Full Source Code : https://github.com/vipul-kumar24/spring-batch-demo/tree/SpringBatch-with-Multithreading

6. Java Builder Design Pattern

- Builder pattern is a creational design pattern.
- It typically solve problem in object oriented programming i.e determining what constructor to use.
- Builder pattern is used to create instance of very complex object having telescoping constructor in easiest way.
- STEPS :
    - Create static inner class into the pojo, why static because we want to return/use current object.
    - Add same fields to it from pojo.
    - Also add the empty constructor and setter of each filed with return type of Builder class.
    - And last but not least add method “build()” which will return the new Person object instance.
- Reference : https://medium.com/@ajinkyabadve/builder-design-patterns-in-java-1ffb12648850
- Example

Computer.java

```
package com.BuilderPattern;

public class Computer {
   private String HDD;
   private String RAM;
   private boolean isGraphicsCardEnabled;
   private boolean isBluetoothEnabled;
   private String brand;

   public String getHDD() {return HDD;}

   public String getRAM() {return RAM;}

   public boolean isGraphicsCardEnabled() {return isGraphicsCardEnabled;}

   public boolean isBluetoothEnabled() { return isBluetoothEnabled; }

   public String getBrand(){ return brand; }

   private Computer(ComputerBuilder builder) {
       this.HDD=builder.HDD;
       this.RAM=builder.RAM;
       this.isGraphicsCardEnabled=builder.isGraphicsCardEnabled;
       this.isBluetoothEnabled=builder.isBluetoothEnabled;
   }
```

```
public static class ComputerBuilder //Inner Class
{
        private String HDD;
        private String RAM;
        private boolean isGraphicsCardEnabled;
        private boolean isBluetoothEnabled;
        private String Brand;

        public ComputerBuilder(String hdd, String ram){
            this.HDD=hdd;
            this.RAM=ram;
        }

       public ComputerBuilder setGraphicsCardEnabled(boolean isGraphicsCardEnabled){
            this.isGraphicsCardEnabled = isGraphicsCardEnabled;
            return this;
        }

        public ComputerBuilder setBluetoothEnabled(boolean isBluetoothEnabled) {
            this.isBluetoothEnabled = isBluetoothEnabled;
            return this;
        }

        public ComputerBuilder setBrand(String brand){
            this.Brand = brand;
            return this;
        }

        public Computer build(){
            return new Computer(this);
        }
    }
}
```

BuilderApplication.java

```
package com.BuilderPattern;
import com.BuilderPattern.Computer.*;

public class BuilderApplication {
    public static void main(String args[]){
        ComputerBuilder computerBuilder = new ComputerBuilder("1 TB","8 GB")
                .setBluetoothEnabled(true)
                .setGraphicsCardEnabled(true);
        Computer computer = computerBuilder.build();
    }
}
```

7. Listener Callback Methods

Example

- MyEventListener.java

```
package com.Listner;

public interface MyEventListener {

   void beforeEvent();
   void afterEvent();
}
```

- MyImplementationClass.java

```
package com.Listner;

public class MyImplementationClass implements MyEventListener {
   @Override
   public void beforeEvent() {
       System.out.println("Performing Before callback from Listener");
   }

   @Override
   public void afterEvent() {
       System.out.println("Performing Before callback from Listener");
   }
}
```

- MyClass.java

```
package com.Listner;

public class MyClass {

   private MyEventListener myEventListener;

   public void someTask(MyImplementationClass
myImplementationClassListener) {
       this.myEventListener = myImplementationClassListener;

       myEventListener.beforeEvent();

       System.out.println("Doing some task");

       myEventListener.afterEvent();
```

```
    }

    public static void main(String[] args) {
        MyClass myClass = new MyClass();
        myClass.someTask(new MyImplementationClass());
    }
}
```