Welcome

Spring Framework

ph no: 9513216462 | email : info@emexotechnologies.com | website: https://www.emexotechnologies.com

Spring Batch

What is Spring Batch?

- Spring Batch is an open source framework for batch processing.
- It provides reusable functions that you can use to process large volumes of records.
- Spring Batch works by first reading items from a specified source. Then in a next step, the items are processed. And finally, the processed items are written to a specified destination.

Why is Spring Batch Useful?

- The requirements of typical batch programs are very similar:
- Restartability:

It is normally required to restart a batch program from where it had failed

- Different readers and writers:

You need the ability to talk to different kinds of data sources and sinks. These include talking to a database, an external messaging service such as JMS, and many others.

Why is Spring Batch Useful? (Contd..)

- **Chunk Processing:**

  If for instance there are a million records to be written to storage, it is a practical idea to split it into manageable chunks of 1000 records each, and write these chinks one at a time. Even if one such chink fails, the other operations are not affected.

- **Ease Of Transaction Management:**

  Transaction management should be simple to implement properly, even when using chunk processing.

- **Ease of parallel processing:**

  It should be possible to run the batch tasks using parallel processing. For this, it is important that the configuration be simple, so that overhead is minimized.

Understanding Spring Batch

- A Job in Spring Batch is nothing but a sequence of Steps. Each Step can be configured before execution, with the following attributes:

  next: The next Step to execute

  tasklet: The task that needs to be done as part of this Step.

  decision: This decides in which situations this Step needs to be run

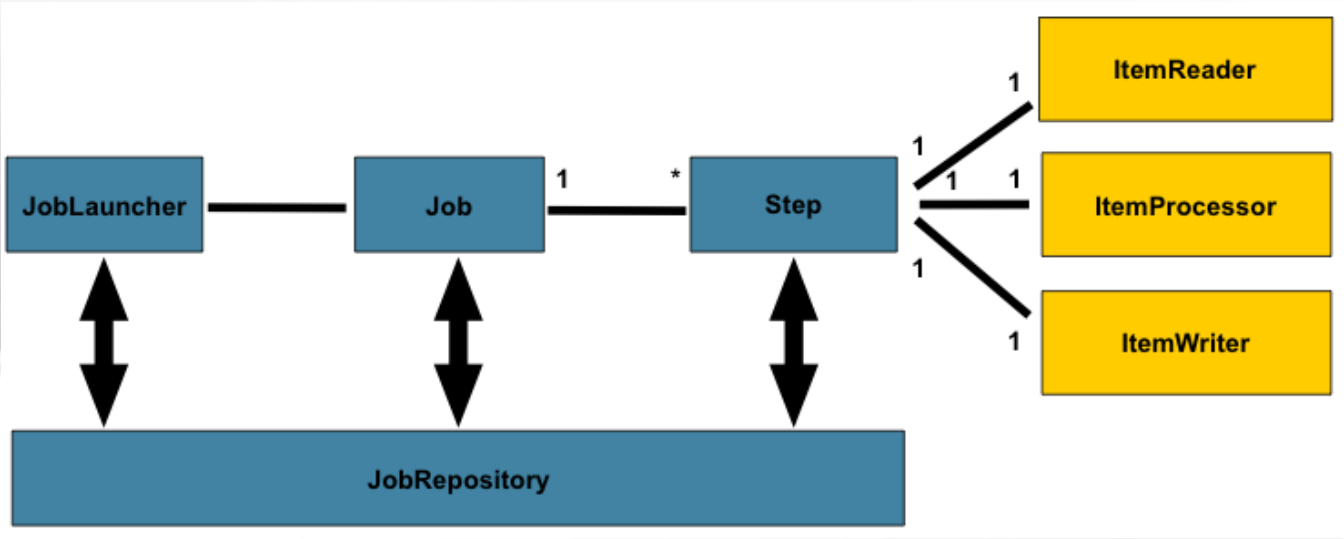
- The Job Instance

A Job Launcher is used in order to execute a Spring Batch Job. Note the following points about a Job thus created:

Each execution of a Job is called a Job Instance. Each Job Instance is provided with a unique execution id, which is useful to restart the job if it fails. A Job can be configured with parameters. These can be passed to it from the Job Launcher.

How Spring Batch Works

- A typical Job would look like the following:
- Each Job can have multiple Steps, and sometimes it is useful to organize the Steps into Flows. Different flows can usually be run in parallel, and the rest of the steps are run in strict sequential order.



Spring Batch Dependency

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-batch</artifactId>
  </dependency>

  <dependency>
        <groupId>org.springframework.batch</groupId>
        <artifactId>spring-batch-test</artifactId>
  </dependency>
```

Spring Batch Annotations

- **@EnableBatchProcessing**

  The @EnableBatchProcessing annotation enables Spring Batch features and provides a base configuration for setting up batch jobs.

- **@Scheduled**

  @Scheduled annotation to the method that executes the job itself.

  Scheduling can be configured with delay, rates, or cron expressions

@EnableBatchProcessing

```
@EnableBatchProcessing
@SpringBootApplication
public class BatchApplication {

    public static void main(String[] args) {
        prepareTestData(1000);
        SpringApplication.run(BatchApplication.class, args);
    }
}
```

@Scheduled

```
// run every 5000 msec (i.e., every 5 secs)
@Scheduled(fixedRate = 5000)
    public void run() throws Exception {
        JobExecution execution = jobLauncher.run(
                customerReportJob(),
                new JobParametersBuilder().toJobParameters()
        );
    }
}
```

Sample Job

- **job that is configured by CustomerReportJobConfig with an injected JobBuilderFactory and StepBuilderFactory**

```
@Configuration
public class CustomerReportJobConfig {
    @Autowired
    private JobBuilderFactory jobBuilders;

    @Autowired
    private StepBuilderFactory stepBuilders;

    @Bean
    public Job customerReportJob() {
        return jobBuilders.get("customerReportJob")
                .start(taskletStep())
                .next(chunkStep())
                .build();
    }

    @Bean
    public Step taskletStep() {
        return stepBuilders.get("taskletStep")
                .tasklet(tasklet())
                .build();
    }

    @Bean
    public Tasklet tasklet() {
        return (contribution, chunkContext) -> {
            return RepeatStatus.FINISHED;
        };
    }
}
```

Main approaches to building a step

- **Tasklet-based**
- **Chunk-oriented processing**

Tasklet Based

- **A Tasklet supports a simple interface that has only one method, execute(), which is called repeatedly until it either returns RepeatStatus.FINISHED or throws an exception to signal a failure.**
- **Each call to the Tasklet is wrapped in a transaction.**

Chunk Oriented Processing

- chunk-oriented processing, refers to reading the data sequentially and creating “chunks” that will be written out within a transaction boundary.
- Each individual item is read in from an ItemReader, handed to an ItemProcessor, and aggregated.
- Once the number of items read equals the commit interval, the entire chunk is written out via the ItemWriter, and then the transaction is committed.

Chunk Oriented Processing Example

```
@Bean
public Job customerReportJob() {
        return
jobBuilders.get("customerReportJob")
        .start(taskletStep())
        .next(chunkStep())
        .build();
    }
```

```
@Bean
public Step chunkStep() {
        return stepBuilders.get("chunkStep")
        .<Customer, Customer>chunk(20)
        .reader(reader())
        .processor(processor())
        .writer(writer())
        .build();
    }
```

ItemReader

- In order to read a list of customers from an XML file, we need to provide an implementation of the interface org.springframework.batch.item.ItemReader
- An ItemReader provides the data and is expected to be stateful. It is typically called multiple times for each batch, with each call to read() returning the next value and finally returning null when all input data has been exhausted.
- Spring Batch provides some out-of-the-box implementations of ItemReader, which can be used for a variety of purposes such as reading collections, files, integrating JMS and JDBC as well as multiple sources, and so on

ItemReader (Contd..)

```
public interface ItemReader<T> {
    T read() throws Exception, UnexpectedInputException,
ParseException, NonTransientResourceException;
}
```

ItemReader (Contd..)

```
public class CustomerItemReader
implements ItemReader<Customer> {
    private final String filename;
    private ItemReader<Customer> delegate;
    public CustomerItemReader(final String
filename) {
        this.filename = filename;
    }
    @Override
    public Customer read() throws Exception
{
        if (delegate == null) {
            delegate = new
IteratorItemReader<>(customers());
        }
        return delegate.read();
    }
    private List<Customer> customers() throws
FileNotFoundException {
        try (XMLDecoder decoder = new
XMLDecoder(new
FileInputStream(filename))) {
            return (List<Customer>)
decoder.readObject();
        }
    }
}
```

ItemProcessor

- **ItemProcessors transform input items and introduce business logic in an item-oriented processing scenario.**
- **They must provide an implementation of the interface org.springframework.batch.item.ItemProcessor:**

```
public interface ItemProcessor<I, O> {
        O process(I item) throws Exception;
}
```

ItemProcessor (Contd..)

- The method process() accepts one instance of the I class and may or may not return an instance of the same type.
- Returning null indicates that the item should not continue to be processed.
- As usual, Spring provides few standard processors, such as CompositeItemProcessor that passes the item through a sequence of injected ItemProcessors and a ValidatingItemProcessor that validates input.

ItemProcessor (Contd..)

```
public class BirthdayFilterProcessor implements
ItemProcessor<Customer, Customer> {

    @Override

    public Customer process(final Customer item) throws Exception {

        if (new GregorianCalendar().get(Calendar.MONTH) ==
item.getBirthday().get(Calendar.MONTH)) {

            return item;

        }

        return null;

    }

}
```

ItemWriter

- **For outputting the data, Spring Batch provides the interface org.springframework.batch.item.ItemWriter for serializing objects as necessary:**

```
public interface ItemWriter<T> {
        void write(List<? extends T> items) throws Exception;
}
```

ItemWriter (Contd..)

- The write() method is responsible for making sure that any internal buffers are flushed.
- If a transaction is active, it will also usually be necessary to discard the output on a subsequent rollback.
- The resource to which the writer is sending data should normally be able to handle this itself.
- There are standard implementations such as CompositeItemWriter, JdbcBatchItemWriter, JmsItemWriter, JpaItemWriter, SimpleMailMessageItemWriter, and others.

ItemWriter(Contd..)

```
public class CustomerItemWriter
implements ItemWriter<Customer>,
Closeable {
    private final PrintWriter writer;

    public CustomerItemWriter() {
        OutputStream out;
        try {
            out = new
FileOutputStream("output.txt");
        } catch (FileNotFoundException e) {
            out = System.out;
        }
        this.writer = new PrintWriter(out);
    }
```

```
    @Override
    public void write(final List<? extends
Customer> items) throws Exception {
        for (Customer item : items) {
            writer.println(item.toString());
        }
    }

    @PreDestroy
    @Override
    public void close() throws IOException {
        writer.close();
    }
}
```

Scheduling Spring Batch Jobs

- By default, Spring Batch executes all jobs it can find (i.e., that are configured as in CustomerReportJobConfig) at startup. To change this behavior, disable job execution at startup by adding the following property to application.properties:
- spring.batch.job.enabled=false
- The actual scheduling is then achieved by adding the @EnableScheduling annotation to a configuration class and the @Scheduled annotation to the method that executes the job itself. Scheduling can be configured with delay, rates, or cron expressions:

Scheduling Spring Batch Jobs (Contd..)

```
@Scheduled(fixedRate = 5000)
public void run() throws Exception {
    jobLauncher.run(
            customerReportJob(),
            new JobParametersBuilder().addLong("uniqueness",
System.nanoTime()).toJobParameters()
    );
}
```


eMexo
TECHNOLOGIES
CALL AT : 09513216462
amazon web services
Dev
Ops
Microsoft Azure
Scala
APACHE Spark
BLOCKCHAIN
UiPath
python
ANGULARJS
hadoop
Data Science
eMexo
TECHNOLOGIES
www.emexotechnologies.com

THANK YOU!

Any questions?

You can find us at

- Email: info@emexotechnologies.com
- Call/WhatsApp: +91 9513216462