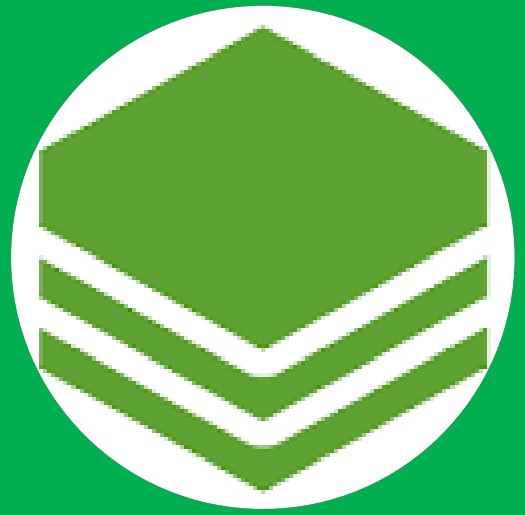


Spring Batch

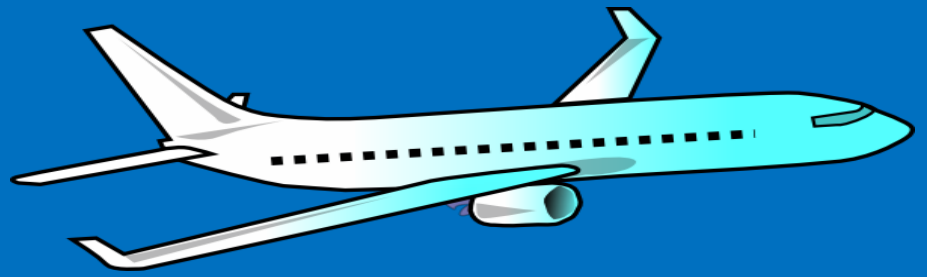
A lightweight, comprehensive batch framework designed to enable the development of robust batch applications vital for the daily operations of enterprise systems.

Efficient Use of Resources

LEARNING
SPRING BATCH

by Pratap Kumar

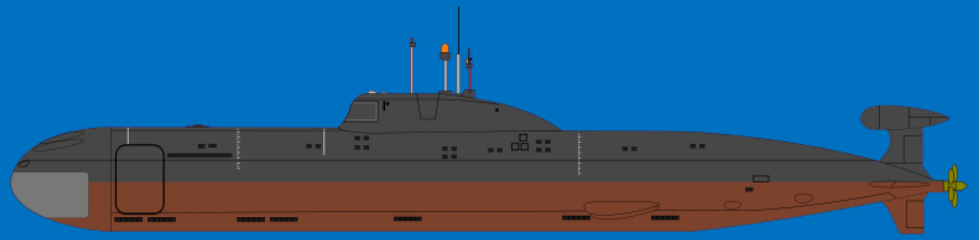
WHAT IS THIS ALL ABOUT ?



OVERVIEW OF BATCH PROCESSING

by Pratap Kumar

TAKE A DEEP DIVE



WHAT IS BATCH PROCESSING?

“Batch Processing…is defined as the processing of a finite set of data without interaction or interruption

FINITE

WITHOUT INTERACTION OR INTERRUPTION

BATCH
USE CASES

1. *ETL*
2. *REPORTING*
3. *DATA SCIENCE*
4. *BIG DATA*
5. *NON-INTERACTIVE PROCESSING*

1.0 RELEASED IN

2008

CURRENT VERSION
4.X


Feature


Videos
Animations
Clip Art
Colors & Shadow
Something New?

1 JOB FLOW STATE MACHINE

2 TRANSACTION HANDLING

3 DECLARATIVE I/0

4 ROBUST ERROR HANDLING

5 SCALABILITY OPTIONS

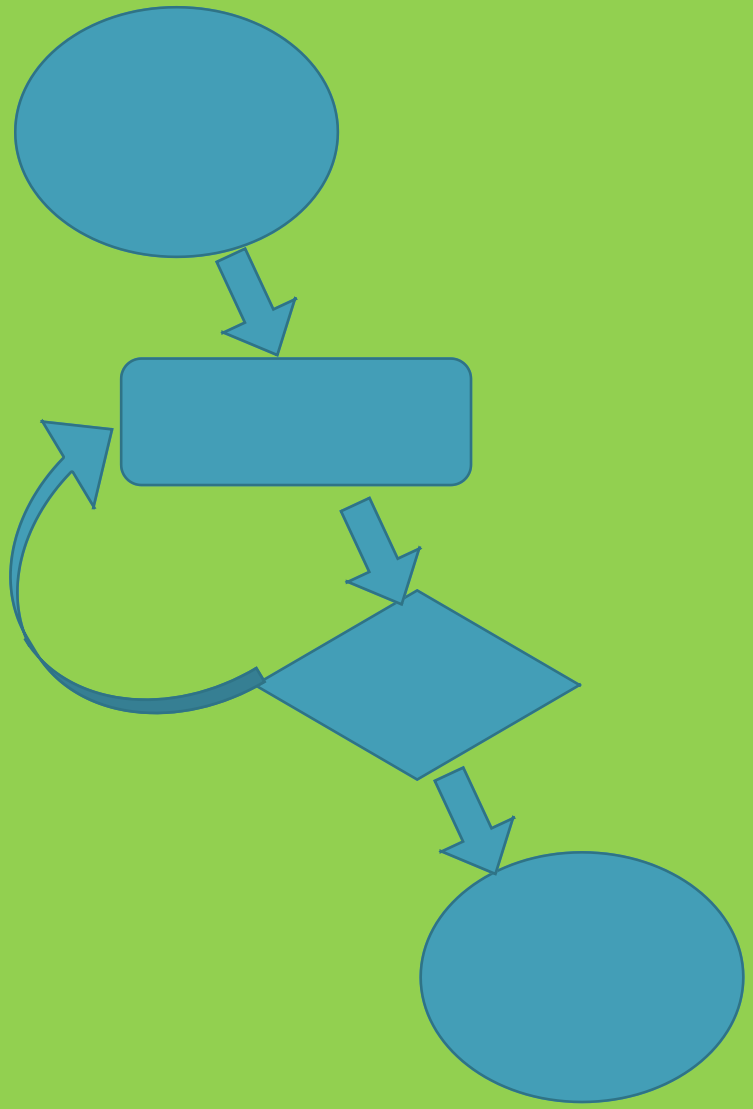
6 BATTLE TESTED

7 BUILT ON SPRING!

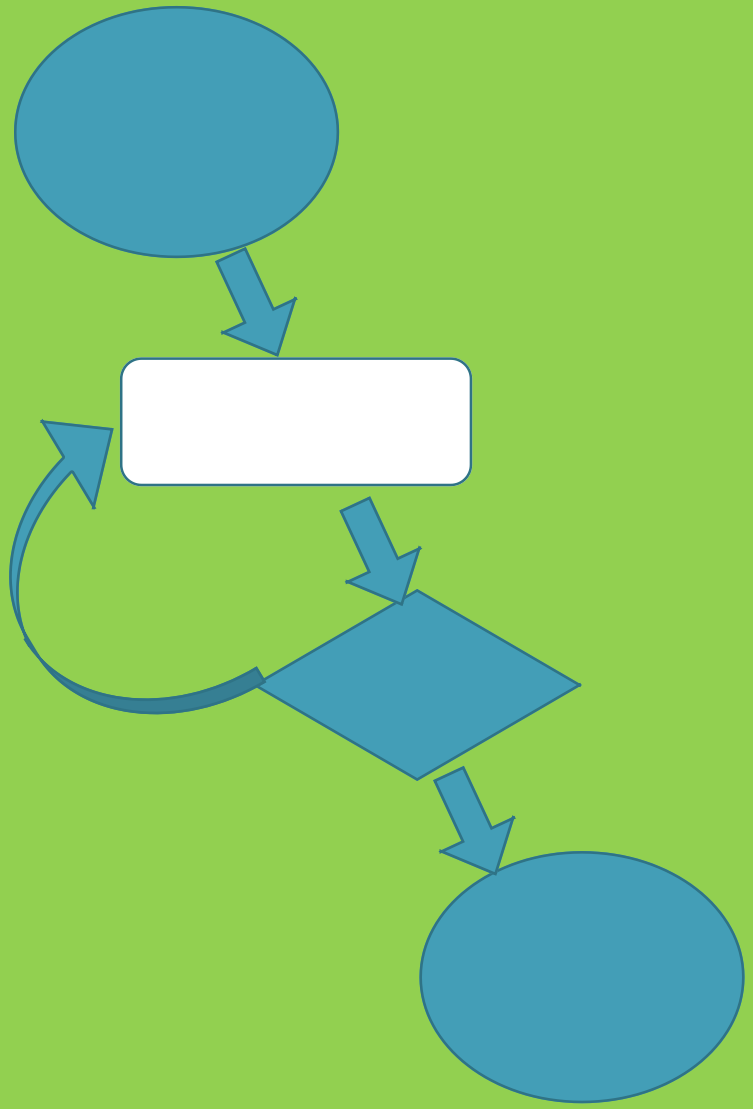
PROJECT SETUP

SPRING BOOT PROJECT

CREATING A SIMPLE JOB

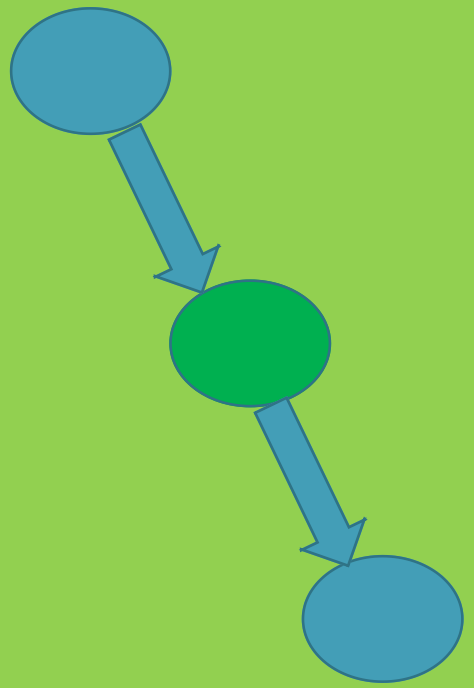


JOB

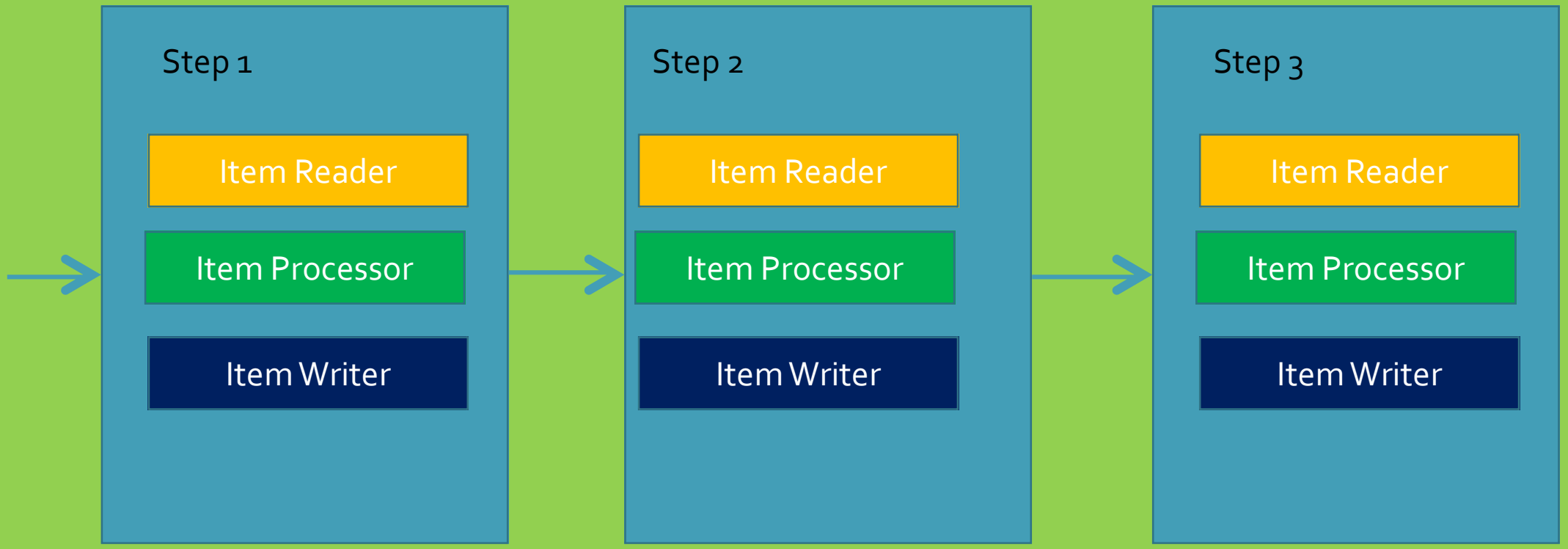


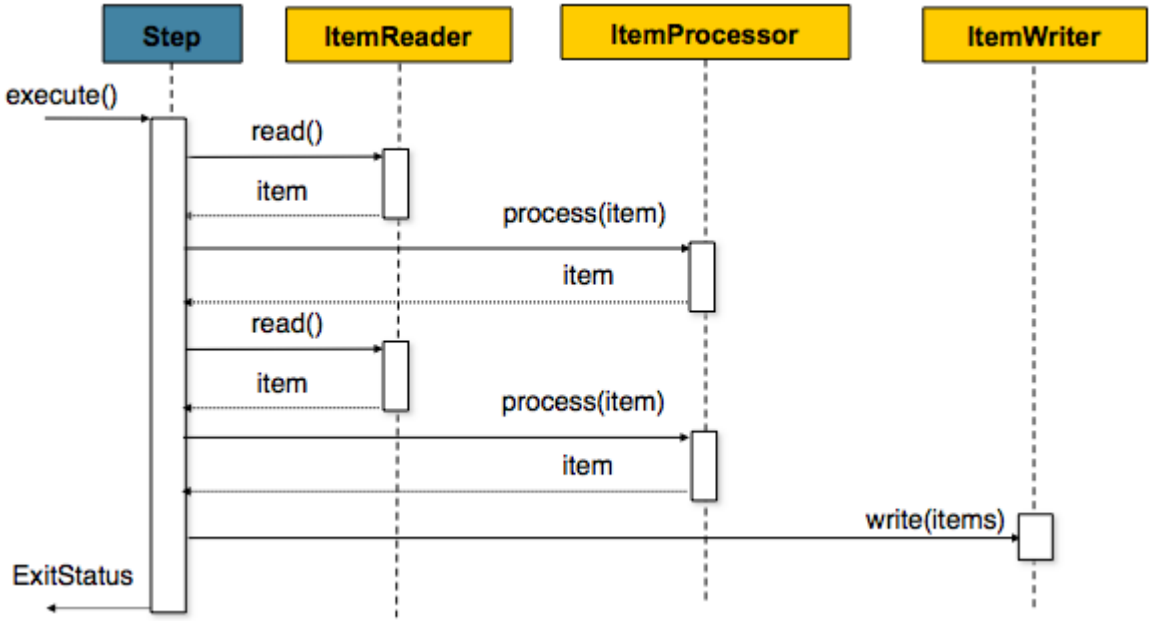
STEP

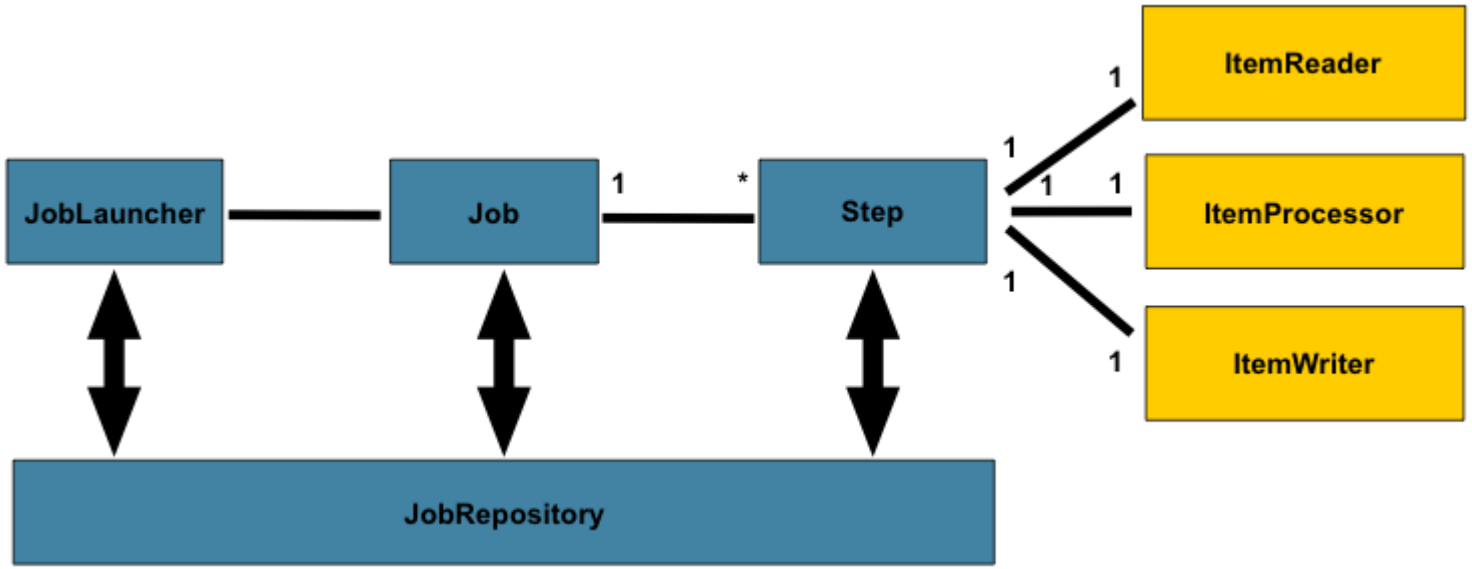
TASKLET

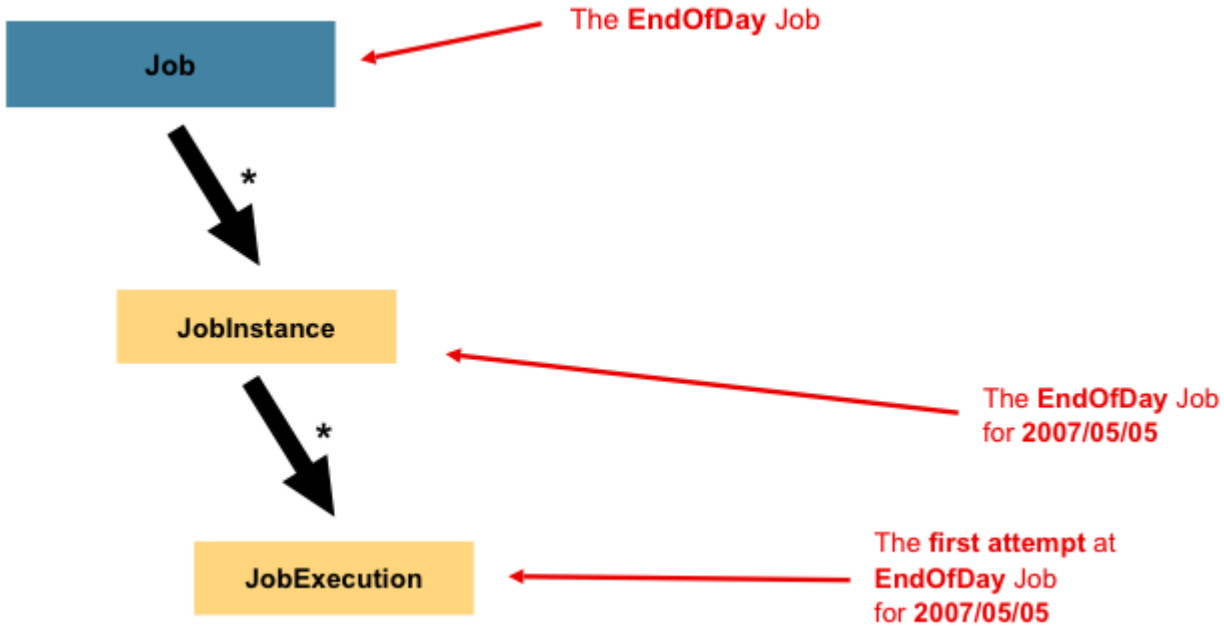


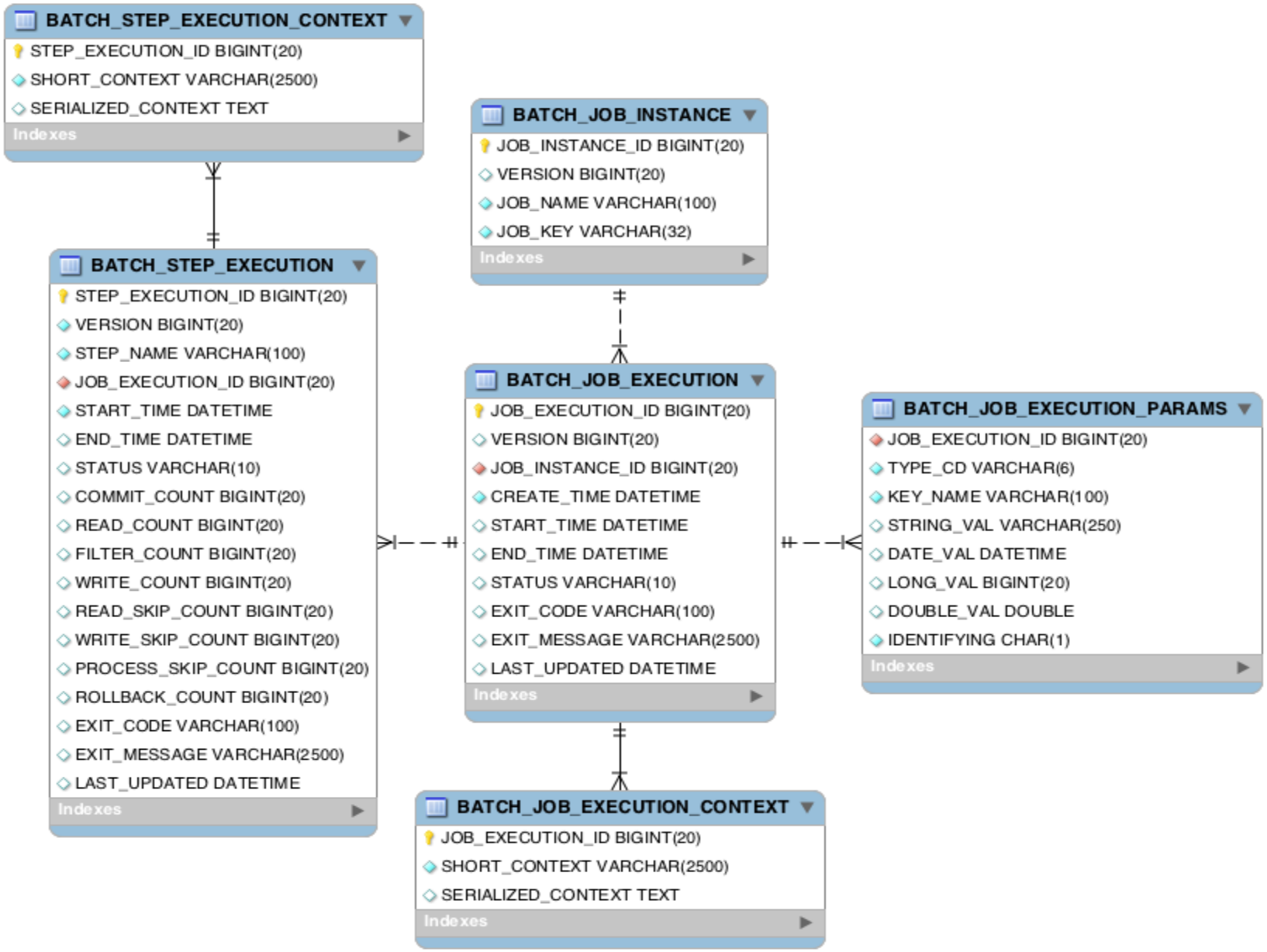
CHUNK

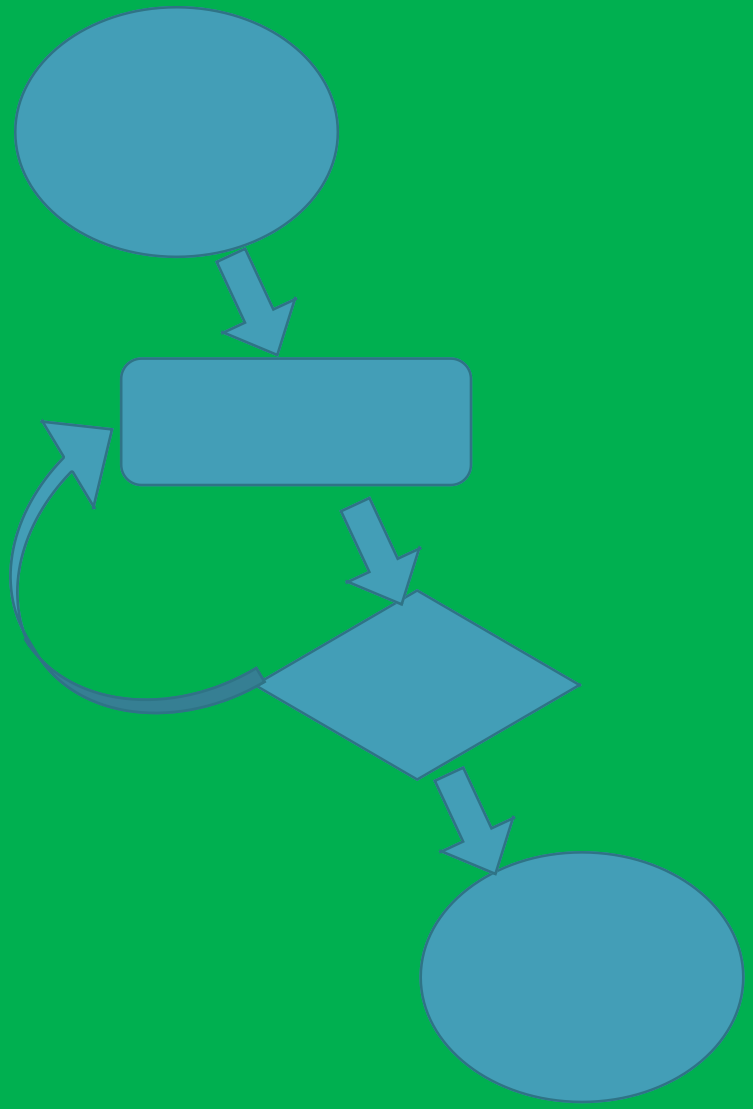

Step 1
Item Reader
Item Processor
Item Writer
Step 2
Item Reader
Item Processor
Item Writer
Step 3
Item Reader
Item Processor
Item Writer


Step
ItemReader
ItemProcessor
ItemWriter
execute()
read()
item
process(item)
item
read()
item
process(item)
item
write(items)
ExitStatus


JobLauncher
Job
1
*
Step
1
1
1
1
1
1
ItemReader
ItemProcessor
ItemWriter
JobRepository


Job
The EndOfDay Job
*
JobInstance
The EndOfDay Job
for 2007/05/05
*
JobExecution
The first attempt at
EndOfDay Job
for 2007/05/05


BATCH_STEP_EXECUTION_CONTEXT
STEP_EXECUTION_ID BIGINT(20)
SHORT_CONTEXT VARCHAR(2500)
SERIALIZED_CONTEXT TEXT
Indexes
BATCH_JOB_INSTANCE
JOB_INSTANCE_ID BIGINT(20)
VERSION BIGINT(20)
JOB_NAME VARCHAR(100)
JOB_KEY VARCHAR(32)
Indexes
BATCH_STEP_EXECUTION
STEP_EXECUTION_ID BIGINT(20)
VERSION BIGINT(20)
STEP_NAME VARCHAR(100)
JOB_EXECUTION_ID BIGINT(20)
START_TIME DATETIME
END_TIME DATETIME
STATUS VARCHAR(10)
COMMIT_COUNT BIGINT(20)
READ_COUNT BIGINT(20)
FILTER_COUNT BIGINT(20)
WRITE_COUNT BIGINT(20)
READ_SKIP_COUNT BIGINT(20)
WRITE_SKIP_COUNT BIGINT(20)
PROCESS_SKIP_COUNT BIGINT(20)
ROLLBACK_COUNT BIGINT(20)
EXIT_CODE VARCHAR(100)
EXIT_MESSAGE VARCHAR(2500)
LAST_UPDATED DATETIME
Indexes
BATCH_JOB_EXECUTION
JOB_EXECUTION_ID BIGINT(20)
VERSION BIGINT(20)
JOB_INSTANCE_ID BIGINT(20)
CREATE_TIME DATETIME
START_TIME DATETIME
END_TIME DATETIME
STATUS VARCHAR(10)
EXIT_CODE VARCHAR(100)
EXIT_MESSAGE VARCHAR(2500)
LAST_UPDATED DATETIME
Indexes
BATCH_JOB_EXECUTION_PARAMS
JOB_EXECUTION_ID BIGINT(20)
TYPE_CD VARCHAR(6)
KEY_NAME VARCHAR(100)
STRING_VAL VARCHAR(250)
DATE_VAL DATETIME
LONG_VAL BIGINT(20)
DOUBLE_VAL DOUBLE
IDENTIFYING CHAR(1)
Indexes
BATCH_JOB_EXECUTION_CONTEXT
JOB_EXECUTION_ID BIGINT(20)
SHORT_CONTEXT VARCHAR(2500)
SERIALIZED_CONTEXT TEXT
Indexes



JOB FLOW

Transitions

```
@Bean
public Job transitionJobSimpleNext(){
        return jobBuilderFactory.get("transitionJobNext")
                .start(step1())
                .next(step2())
                .next(step3()).build();
}
@Bean
public Job transitionJobSimpleNext(){
        return jobBuilderFactory.get("transitionJobNext")
                .start(step1())
                .on("COMPLETED").to(step2())
                .from(step2()).on("COMPLETED").to(step3())
                .from(step3()).end()
                .build();
}
```

Transitions

Spring batch provided terminal state fail() , stop(), stopAndRestart(step3) , end()

Spring batch support 2 different status code

batch status
that is a finite number of status the spring batch the framework understand

exist status/code
used to detect what transition to execute next

Flow

A flow is nothing but a collection steps and the related transitions

Spring batch allows to create our own flow explicitly to use them as the reusable components.


```
@Bean
public Flow foo() {
        FlowBuilder<Flow> flowBuilder = new FlowBuilder<>("foo");
        flowBuilder.start(step1())
                .next(step2())
                .end();
        return flowBuilder.build();
}
```

FlowFirstConfiguration.java

```
@Bean
public Job flowFirstJob(Flow flow) {
        return jobBuilderFactory.get("flowfirstjob5")
                        .start(flow)
                        .next(myStep1())
                        .end()
                        .build();
}
```

FlowLastConfiguration.java

```
@Bean
public Job flowLastJob(Flow flow) {
        return jobBuilderFactory.get("flowlastjob5").
                        start(myStep2()).
                        on("COMPLETED").
                        to(flow)
                        .end()
                        .build();

}
```

Split

Executing multiple flow in parallel

```
@Bean
public Job job() {
        return jobBuilderFactory.get("job")
        .start(flow1())
        .split(new SimpleAsyncTaskExecutor())    // parallalize the flow
        .add(flow2())
        .end()
        .build();
}
```

Decisions

JobExecutionDecider

Nested Job

- Spring batch has the capabilities of composing job and flows that are really as complex as you see fit.
- For more complex flow composition may provide better model than placing awful logic in single complex job.
- Spring batch allows for that by allowing one job to execute another, this is accomplished by job-step.
- Job step launch another job within the scope of a step
- By default it launch within the same JVM and the same execution thread as the current job.

Listener

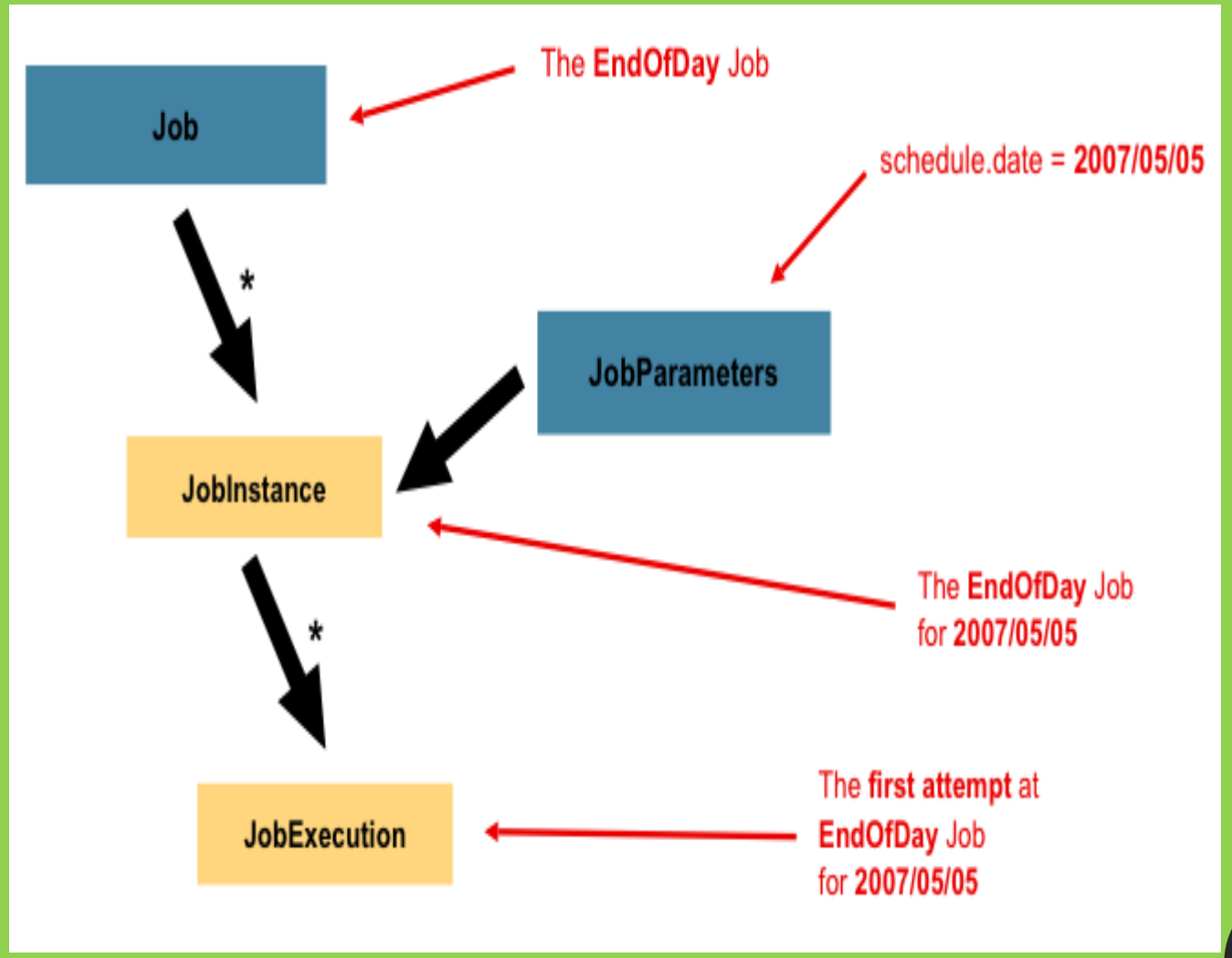
- Spring batch provides ability to handle complex processing across different use cases.
- In some use cases it can be helpful to insert logic in given point in the job lifecycle.
- Listener provided by the spring batch allow a developer to take control of the execution flow at just about any junction point within a given batch job
- Listener functionality implemented by implementing appropriate interface and then adding that impl to your job or steps configuration.
- each of the interface has a support class for execution implementation

Listener

- JobExecutionLIstener
- StepExecutionLIstener
- ChunkLIstener
- ItemReadListener
- ItemWriteListener
- ItemProcessListener

- Annotation
  - org.springframework.batch.core.annotation

Job Parameters



Job Parameters

ItemReader

- One of the key feature of the spring batch is the versatility of the of the out of the box ItemReader and ItemWriter available.
- ItemReader is The core interface used for the step input
- When used chunk based step the ItemReader is responsible to provide input to the step
- The ItemReader interface have a single method read()
- The read() method return a single item , this is essentially the individual unit of processing that occur over and over.

```
public class StatelessItemReader implements ItemReader<String>
{
        private final Iterator<String> data;

        public StatelessItemReader(List<String> data) {
                this.data = data.iterator();
        }

        public String read() throws Exception {
                if (this.data.hasNext()) {
                        return this.data.next();
                } else {
                        return null;
                  }
        }
}
```

ItemReader Database

- In the enterprise , one of the common source of data is the Relational database
- Spring batch provides few different option for reading data from the database each with their own pros and cons.
- Two of the common options are
  - JdbcCursorItemReader
  - JdbcPagingItemReader
  - HibernateCursorItemReader
  - HibernatePagingItemReader
  - JpaPagingItemReader
  - StoredProcedureItemReader

JdbcCursorItemReader

```
@Bean
public JdbcCursorItemReader<Account> cursorItemReader() {

        JdbcCursorItemReader<Account> reader =
                        new JdbcCursorItemReader<>();

        reader.setSql("SELECT ACCID, NAME, BALANCE
                                FROM ACCOUNT ORDER BY ACCID");
        reader.setDataSource(dataSource);
        reader.setRowMapper(new AccountRowMapper());

        return reader;
}
```

it is simple
it is state full
Not thread safe

JdbcPagingItemReader

```
@Bean
public JdbcPagingItemReader<Account> pagingItemReader() {
        JdbcPagingItemReader<Account> reader = new JdbcPagingItemReader<>();
        reader.setDataSource(dataSource);
        reader.setFetchSize(10);
        reader.setRowMapper(new AccountRowMapper());

        OraclePagingQueryProvider queryProvider = new OraclePagingQueryProvider();
        queryProvider.setSelectClause("accid , name , balance");
        queryProvider.setFromClause("from account");

        Map<String, Order> sortKeys = new HashMap<>();
        sortKeys.put("accid", Order.ASCENDING);

        queryProvider.setSortKeys(sortKeys);
        reader.setQueryProvider(queryProvider);

        return reader;
}
```

is thread safe

ItemReader
Flat file

- FlatFileItemReader

```
@Bean
public FlatFileItemReader<Account> accountItemReader() {
        FlatFileItemReader<Account> reader  = new FlatFileItemReader<>();
        reader.setLinesToSkip(1);
        reader.setResource(new ClassPathResource("account.csv"));

        DefaultLineMapper<Account> customerLineMapper =
                                                new DefaultLineMapper<>();
        DelimitedLineTokenizer tokenizer = new DelimitedLineTokenizer();
        tokenizer.setNames("accid","name" ,"balance");

        customerLineMapper.setLineTokenizer(tokenizer);
        customerLineMapper.setFieldSetMapper(new CustomerFieldSetMapper());

        customerLineMapper.afterPropertiesSet();

        reader.setLineMapper(customerLineMapper);
        return reader;
}
```

ItemReader XML

- Spring batch provide an ItemReader that handles xml in a very robust way.
- It is called StaxEventItemReader
- When using StaxEventItemReader it is helpful to think of an xml file as a list of chunks
- The StaxEventItemReader will read each chunk as an individual item to be return by the ItemReader read methods

```
<accounts>
        <account>
                <accid>1</accid>
                <name>pratap</name>
                <balance>2000.0</balance>
        </account>
</accounts>
```

ItemReader Multiple Resource

- Batch processing are known for the volume of data they processes
- Handling millions of records is not uncommon scenario in world of batch
- So our framework need to be ready to handle the load
- One of the way the large data set can be delivered to us is through multiple input files
- MultiResourceItemReader is a tool within spring batch that can help in this requirement

Item Writer

- Spring batch has a collection of ItemWriter implementation used to provide ability to generate output of a step.
- When using a chunk based step , the ItemWriter is responsible for providing the output of the step
- The ItemWriter interface has a single method
  - void write(List<? extends T> items) throws Exception

ItemWriter Database

- JdbcBatchItemWriter
- HibernateItemWriter
- JpaItemWriter

ItemWriter
Flat File

- FlatFileItemWriter

ItemWriter Xml

- Spring batch provides facility to write xml in a performant and transactional friendly manner.
- The StaxEventItemWriter provides ability to use any spring marshaller to generate xml chunks to be written for each item.
- This allows the reuses of any xml technologies you are comfortable with in the context of spring batch.

ItemWriter Multiple Destination

- Writing To Multiple Destinations
- Reading from one place and writing to another is a common pattern in batch processing.
- It is not uncommon to write to multiple location in a batch application.
- Spring batch provides a couple of useful utilities to help with this.
- How to write the same output to multiple location as well as selectively write some output to one location and some to other.

Item Processor

- In a chunk based step, The ItemReader is responsible for providing the input, The ItemWriter responsible to generating the output.
- However the ItemProcessor is where the business logic lives
- ItemProcessor <I , O >
  - O process(I item)
  - ItemProcessor
  - Filtering Items
  - Validating Items (Validating Item Processor)
  - Composite Item Processor

Item Processor Formating

```
public class UpperCaseItemProcessor implements
ItemProcessor<Account, Account> {

        public Account process(Account item) throws
Exception {

        return new Account(item.getAccid(),

                item.getName().toUpperCase(),

                item.getBalance());

        }

}
```

Item Processor Filtering Items

```
public class FilteringItemProcessor
implements ItemProcessor<Account, Account> {
public Account process(Account item) throws
Exception {
        if( item.getAccid() % 2 == 0) {
                return null;
                }
        return item;
        }
}
```

Item Processor Validating Items

```
@Bean
public ValidatingItemProcessor<Account> itemProcessor() {
ValidatingItemProcessor<Account>
accountValidatingItemProcessor
= new ValidatingItemProcessor<Account>(new
AccountValidator());
return accountValidatingItemProcessor;
}
```

```
public class AccountValidator implements Validator<Account>{
public void validate(Account acc) throws ValidationException {
if( acc.getName().startsWith("A")) {
throw new ValidationException("first name begin with a , invlaid
data ");
}}}
```

Item Processor CompositeItemProcessors

```
@Bean
public CompositeItemProcessor<Account, Account>
itemProcessor() throws Exception {
List<ItemProcessor<Account, Account>> delegates = new
ArrayList<>();
delegates.add(new FilteringItemProcessor());
delegates.add(new UpperCaseItemProcessor());

CompositeItemProcessor<Account, Account> processor =
new CompositeItemProcessor<>();
processor.setDelegates(delegates);
processor.afterPropertiesSet();

return processor;
}
```

Error Handling Restart

- Deadlock user exception
- Remote service interruption
- Network hiccups

- Spring batch has the retry capabilities in built.
- Using spring retry libraries spring batch allows you to retry the processing or writing of any item in the normal processing lifecycle.
- The way it work is simple, you can figure the exception you want to indicate an error that is retry able.
- If that exception is thrown from a component that is retry able ,specifically ItemProcessor / ItemWriter the current transaction is rolled back and items are tried one by one
- This occurs until item succeed or the retry max reached.
- Spring batch provides a number of ways to customize behavior within this functionalities by providing things like retry policy or backup policy.

Error Handling Restart

- The ability to handle error in robust way is the foundation concept within spring batch.
- Its one of the main reason the framework can be trusted to handle large volume of data it does in mission critical application.
- There are many ways to handling errors in spring batch.
- Restart ability : within a spring batch job , JobRepository is used to maintain the state of the job as it is executed ,
- When something goes wrong this allows the job to pick up where it left off
- By default , if an uncaught exception is thrown spring batch will end the processing of the batch job.
- If the job is is restarted with the same job parameter it will pick up where it left off
- You can configure and override this behavior.

Error Handling
Skip

- One of the error handling facility spring batch provides is the ability to skip item that cause the error .
- The ability to skip item and keep processing is a great way to handle bad data.
- To skip an item within spring batch you configure a step with an exception that indicate that an item should be skipped
- Along with that exception you also let spring batch know how many item is valid to skip

Error Handling
Skip Listeners

- One of the most useful error handling options within spring batch is skip functionalities.
- The ability of spring batch to skip an item that causes an error provides a robust option for developer to address it with.
- However just skipping an item is not just enough, that item in the batch for reason and probably still needs to be process
- The ability to save the item for later evaluation can make the difference.
- Spring batch provide a listener for you to be able to record and react to what happen during the skip operation.