Spring Batch

Created by :
Sangeeta Joshi
sangeeta.joshi@brewtechblue.com
8669568928

Agenda

- Why & What is Spring Batch?
- Domain Architecture
- Configuration
- Listeners
- Item Readers & Writers

Background

- always more focus was on creating frameworks for
  - web-based and microservices-based application development

- a notable *lack of focus on reusable architecture* frameworks to accommodate
  - Java-based *batch processing needs*,
    despite there was a continuous need to handle such processing within enterprise IT environments.

  - *The lack of a standard, reusable batch architecture*
    That resulted in in-house solutions developed within client enterprise IT functions.

- *Spring Source (now Pivotal) and Accenture* collaborated to target and solve this very issue.

Problem: Bulk Processing of business data

Need :- ***Bulk Processing of business data***

Business operations are required to be performed on huge amount of data in Enterprise Applications.

Operations such as :

- time-based events like month-end calculations,
- notices, or correspondence.

  Automated, complex processing of large volumes of information

  (*without user interaction*.)
- insurance benefit determination or rate adjustments (i.e. Periodically applying complex business rules)
- Data that typically requires formatting, validation, and processing in a transactional manner into the system of record.
- *Batch processing* is used to process billions of transactions every day for enterprises.

Solution: Spring Batch

What is Spring Batch ?

- *A Batch Framework to process bulks of business data* (without user interaction)
  - to enable the development of robust batch applications
  - vital for the daily operations of enterprise systems.
  - Lightweight
  - comprehensive

- *It provides reusable functions that are essential*
  - in processing large volumes of records like
    - logging/tracing,
    - transaction management,
    - job processing statistics,
    - job restart, skip, and resource management.
    - features that enable extremely high-volume and high performance batch jobs through optimization and partitioning techniques

Spring Batch

- Spring Batch can be used:
  - *In simple use cases*
    - reading a file into a database
    - running a stored procedure
  - *In complex, high volume use cases*
    - moving high volumes of data between databases
    - transforming it, and so on.
    - to process significant volumes of information in a highly scalable manner

Framework Characteristics

- Productivity
- POJO-based development approach
- Simpler and easier for developers to access more advance enterprise services when necessary.
- Spring Batch is not a scheduling framework.
- It is intended to work in conjunction with a scheduler, not replace a scheduler.

Batch Processing Tasks

A typical batch program generally:

- *Reads* a large number of records from a database, file, or queue.
- *Processes* the data in some fashion.
- *Writes back* data in a modified form.

Batch Processing

- Commit batch process periodically
- Concurrent batch processing: parallel processing of a job
- Enterprise message-driven processing
- Massively parallel batch processing

Batch Processing Tasks

- Manual or scheduled restart after failure
- Sequential processing of dependent steps
- Partial processing: skip records (for example, on rollback)
- Whole-batch transaction, for cases with a small batch size or existing stored procedures/scripts

Technical Objectives

- use the Spring programming model:
  - Lets developers concentrate on business logic and let the *framework take care of infrastructure*.
- *Clear separation of concerns* between
  - the infrastructure
  - the batch execution environment
  - and the batch application.
- Provide common, core execution services as *interfaces* that all projects can implement.

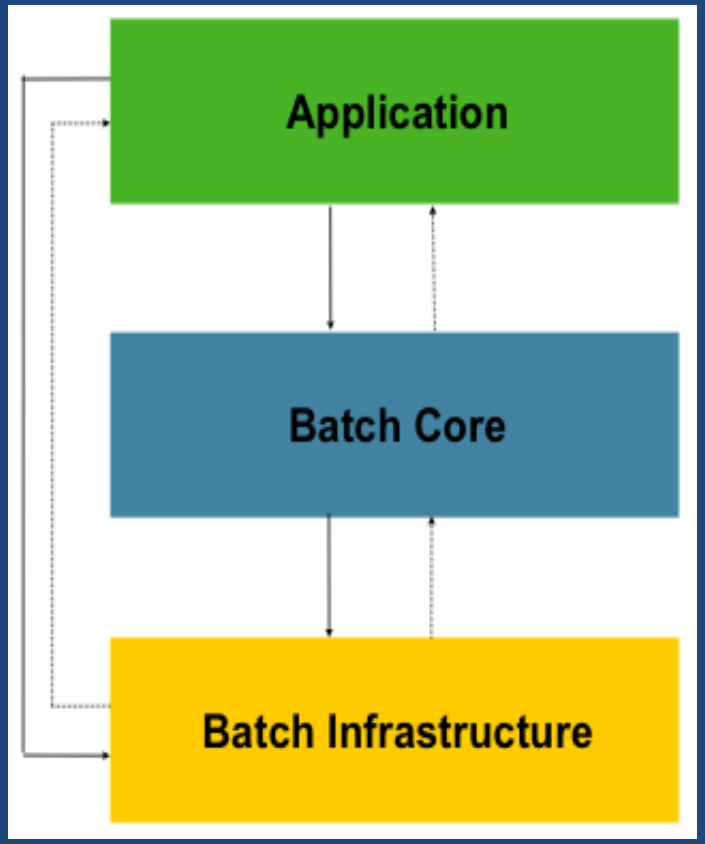
Technical Objectives

- Providing *'out of the box'* default implementations of the core execution interfaces
- Easy to configure, customize, and extend services
- All existing core services should be *easy to replace or extend*, without any impact to the infrastructure layer.
- Provide a *simple deployment model*,
  - with the architecture JARs completely separate from the application,
  - built using Maven.

Spring Batch Architecture

Layered architecture that supports:

- the extensibility
- ease of use for end-user developers.



Spring Batch Architecture

three major high-level components

The application : contains all batch jobs and custom code written by developers

The Batch Core : contains the core runtime classes necessary to launch and control a batch job. For example:

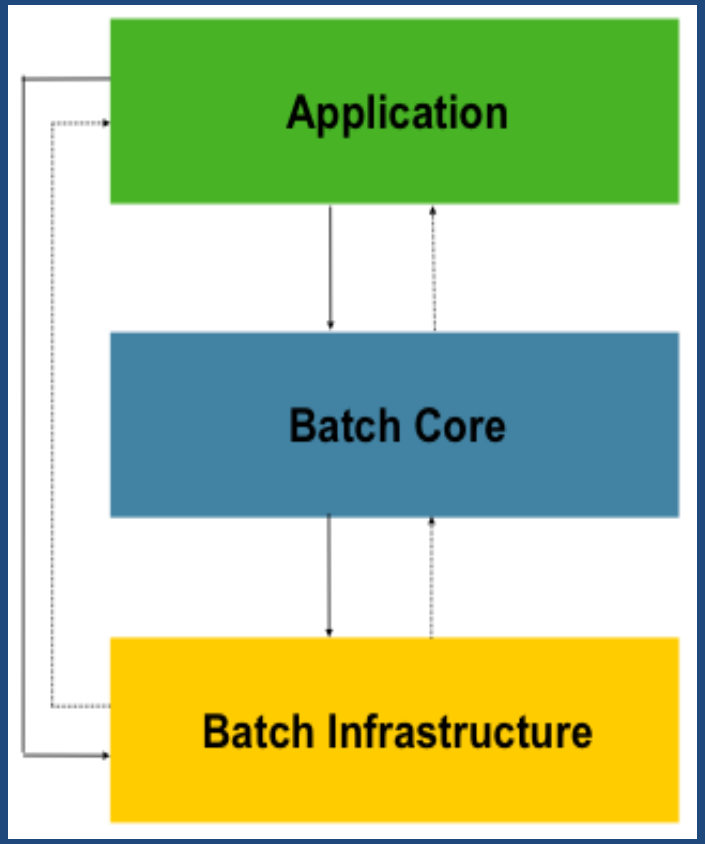
JobLauncher, Job, and Step.

The infrastructure : contains common readers and writers and services which are used both by:

application developers

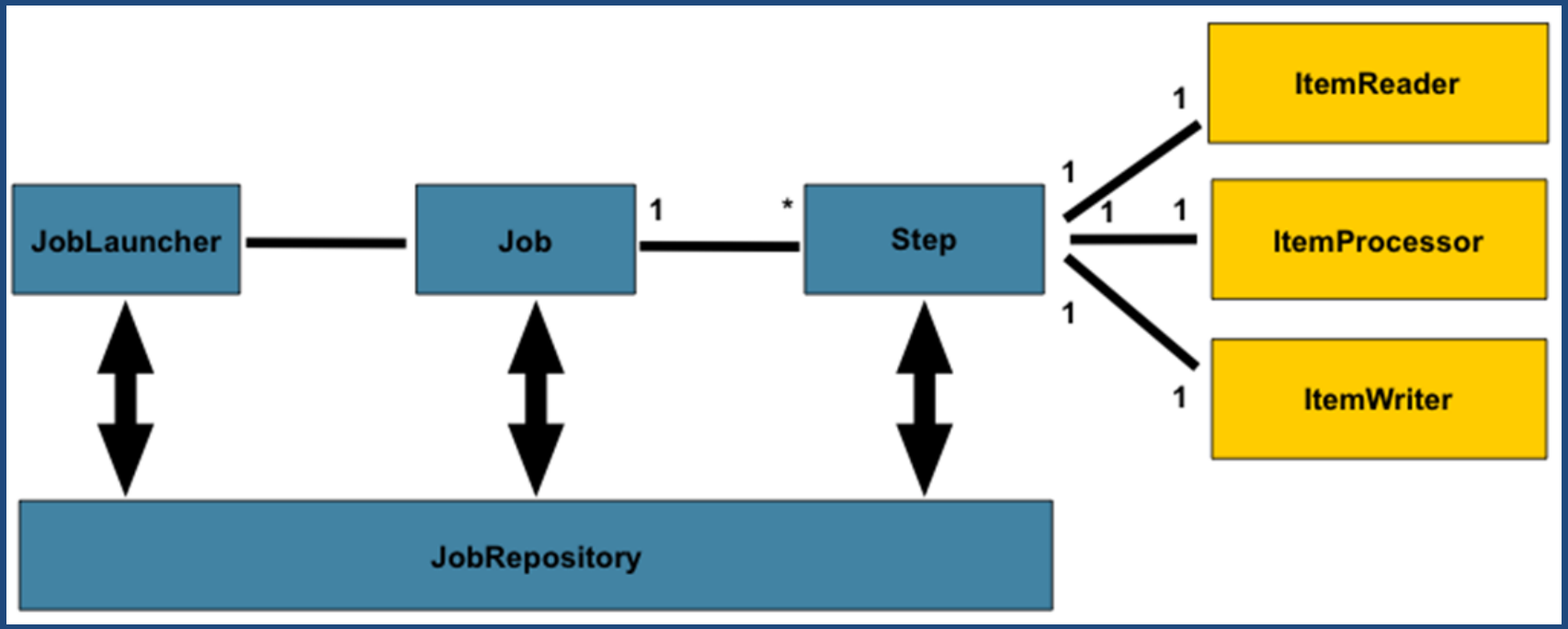
and the core framework itself

Both Application and Core are built on top of a common infrastructure.



Domain Language / Terminology

Batch Stereotypes



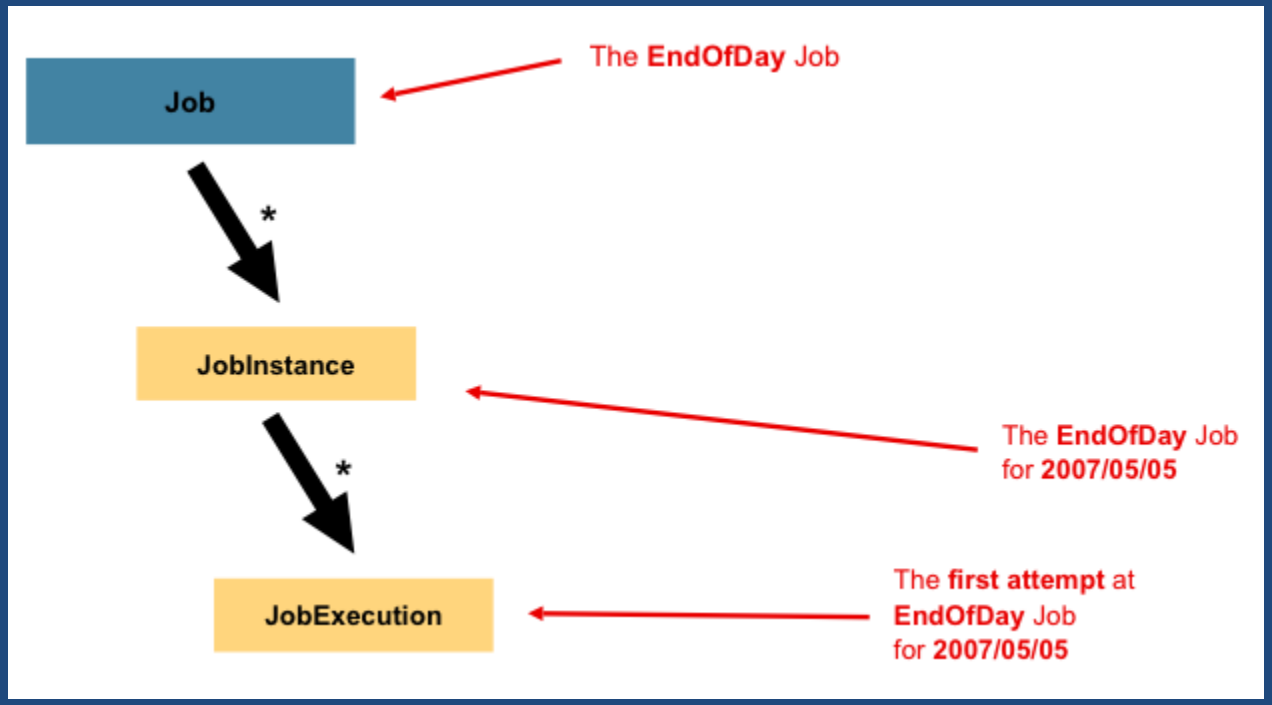
A framework for batch processing – execution of a series of jobs.

- A job consists of many steps
  - a step may read data from a CSV file, process it and write it into the database. Spring Batch provides many Classes to read/write CSV, XML and database.
- and each step consists of a READ-PROCESS-WRITE task
  - For “READ-PROCESS-WRITE” process, it means “read” data from the resources (csv, xml or database), “process” it and “write” it to other resources (csv, xml and database).
- or single operation task (tasklet).
  - it means doing single task only, like clean up the resources after or before a step is started or completed.

For example,

- For “single” operation task (tasklet), and the steps can be chained together to run as a job.
- 1 Job = Many Steps.
- 1 Step = 1 READ-PROCESS-WRITE or 1 Tasklet.
- Job = {Step 1 -> Step 2 -> Step 3} (Chained together)

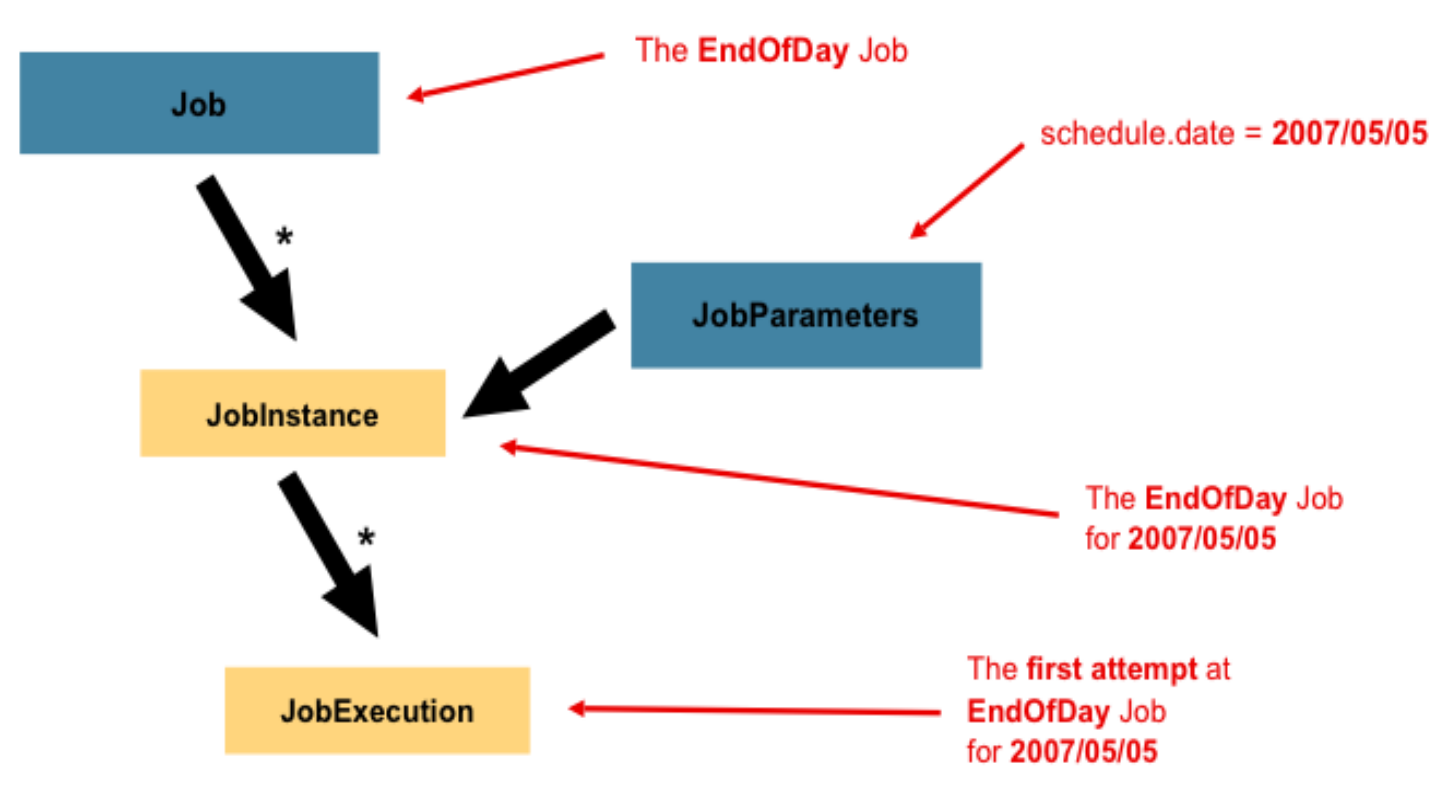
Job


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

Job

- *Job* is an entity that encapsulates an entire batch process
- *Job* is simply a *container for Step* instances.
  - It combines multiple steps that belong logically together in a flow
  - allows for configuration of properties *global to all steps*, such as restartability.
- *SimpleJob*
  - A class that implements Job Interface

Job
Instance, Parameters, Execution



Job
Instance, Parameters, Execution

- A Job :
  - defines what a job is
  - and how it is to be executed,
- A JobInstance:
  - is a purely organizational object to group executions together,
  - Consider a batch job that should be run once at the end of the day, such as the 'EndOfDay' Job
  - There is one 'EndOfDay' job, but *each individual run* of the Job must be tracked separately.
  - In case of this job, there is one logical *JobInstance* per day.

Job
Instance, Parameters, Execution

- A JobExecution:
  - the *primary storage mechanism* for what actually happened during a run
  - contains many more properties that must be controlled and persisted,
  - A JobExecution refers to the technical concept of a single attempt to run a Job.
  - An execution may end in failure or success, but
  - JobInstance corresponding to a given execution is not considered to be complete unless the execution completes successfully.

Job
Instance, Parameters, Execution

- Job Parameters
  - "How is one JobInstance distinguished from another?" : *JobParameters*
  - A JobParameters object holds a set of parameters used to start a batch job.
  - *JobInstance = Job + identifying JobParameters*

Job configuration

- The job configuration contains:
  - The simple name of the job.
  - Definition and ordering of Step instances.
  - Whether or not the job is restartable.
- A Job Configuration can be:
  - an XML configuration file

    or
  - Java-based configuration

XML Configuration

- In a Spring Batch application, we configure :
  - job
  - step
  - JobLauncher
  - JobRepository
  - Transaction Manager
  - readers,
  - writers
- using XML tags provided in *Spring Batch namespace*.
- Therefore, we will include this namespace in our XML file.

Java Configuration-Job

- For java based configuration:
  - *builders* are made available for the instantiation of a Job.

```
@Bean
public Job footballJob() { return
this.jobBuilderFactory.get("footballJob")
.start(playerLoad()) .next(gameLoad())
.next(playerSummarization()) .end() .build(); }
```

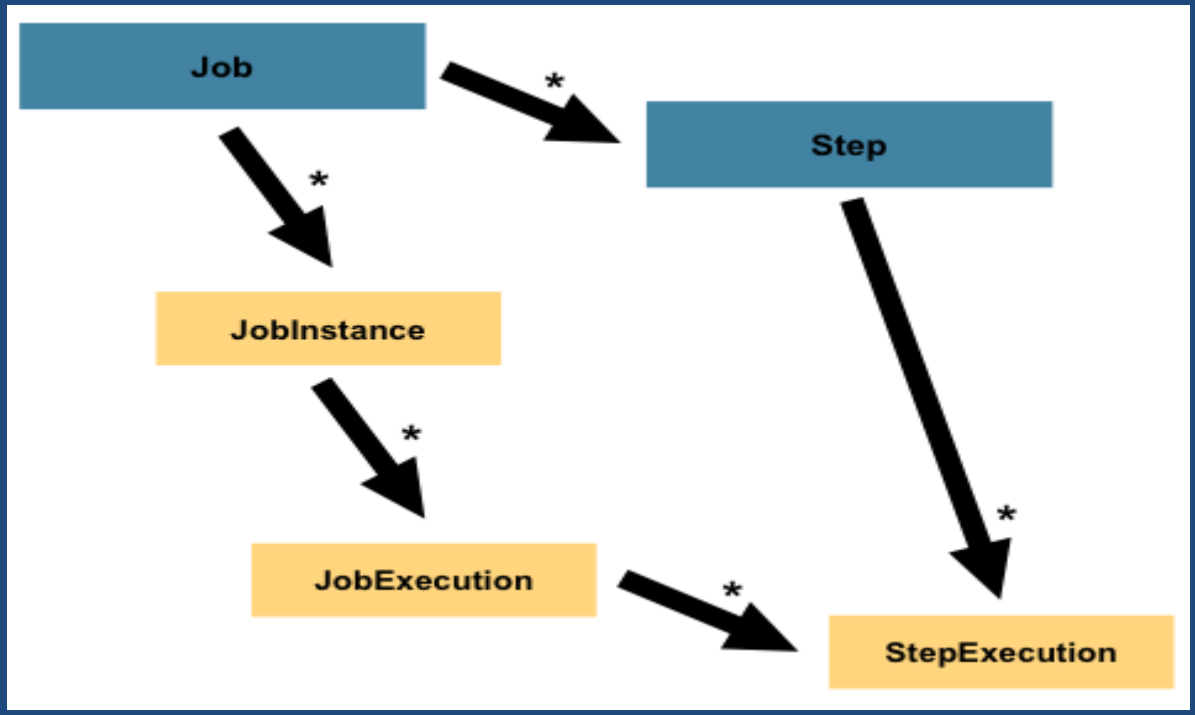
Job

```
@Bean
public Job footballJob() {
    return this.jobBuilderFactory.get("footballJob")
                    .start(playerLoad())
                    .next(gameLoad())
                    .next(playerSummarization())
                    .end()
                    .build();
}
```

Step

- A Step: a domain object that encapsulates
  - *an independent,*
  - *sequential phase of a batch job.*
- Therefore, every Job is composed entirely of *one or more steps*.
- A Step : contains all necessary information *to define and control the actual batch processing*.
- A Step : can be simple or complex.
- A simple Step : might load data from a file into the database.
- A more complex Step : may have complicated business rules to apply as part of the processing.
- As with a Job, a Step has *an individual StepExecution* that correlates with a unique JobExecution

Job Hierarchy With Steps



Step Execution

- StepExecution : represents *a single attempt to execute a Step.*
- A new StepExecution is created each time a Step is run.
- However, if a step fails to execute because the step before it fails, no execution is persisted for it.
- A StepExecution is created only when its Step is actually started.
- Step executions : objects of the *StepExecution class*.
- Each execution contains
  - a reference to its corresponding step
  - A reference to JobExecution
  - transaction related data,
    - such as commit and rollback counts
    - and start and end times.
  - an ExecutionContext,
    - which contains any data needs to have persisted across batch runs,
    - such as statistics or state information needed to restart.

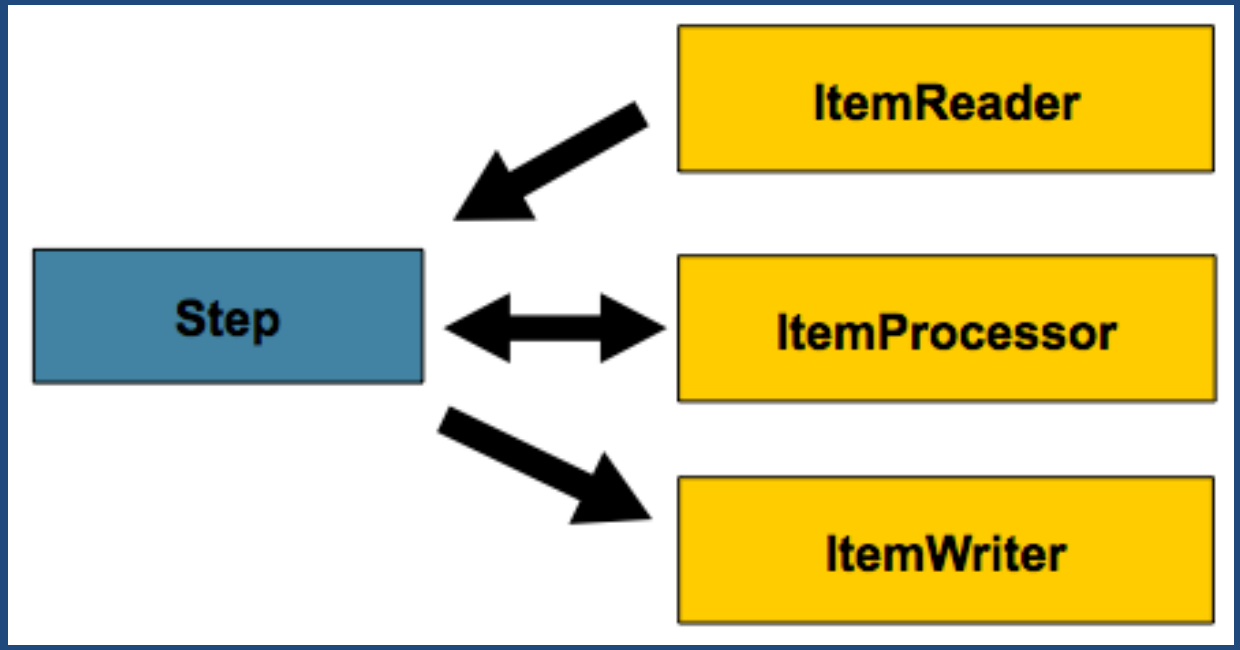
Execution Context

- An ExecutionContext :
  - represents a collection of key/value pairs that are persisted and controlled by the framework
  - in order to allow developers a place to store persistent state that is scoped to a StepExecution object or a Job Execution object.
  - The best usage example is to facilitate restart.
  - Using flat file input as an example, while processing individual lines,
    - the framework periodically persists the ExecutionContext at commit points.
    - Doing so allows the ItemReader to store its state in case a fatal error occurs during the run or even if the power goes out.
    - All that is needed is to put the current number of lines read into the context and the framework will do the rest

```
executionContext.putLong(getKey(LINES_READ_COUNT),reader.getPosition());
```

- there is at least one *ExecutionContext* per JobExecution and one for every StepExecution

Step



Job Repository

- JobRepository:
  - The persistence mechanism for all of the Stereotypes
  - Provides CRUD operations for JobLauncher, Job, and Step implementations.
  - When a Job is first launched:
    - a JobExecution is obtained from the repository,
    - and, during the course of execution, StepExecution and JobExecution implementations are persisted by passing them to the repository.
  - @EnableBatchProcessing : provides a JobRepository as one of the components automatically configured out of the box.

Job Launcher

- JobLauncher : a simple interface for launching a Job with a given set
  of JobParameters.

```
public interface JobLauncher {

   public JobExecution run(Job job, JobParameters params)
         throws  JobExecutionAlreadyRunningException,
                 JobRestartException,
                 JobInstanceAlreadyCompleteException,
                 JobParametersInvalidException;
}
```

Tasklet

- Strategy for processing in a step

  RepeatStatus *execute* (StepContribution contribution,
  ChunkContext chunkContext)
  throws java.lang.Exception

Parameters:

contribution - mutable state to be passed back to update the current step execution

chunkContext - attributes shared between invocations

Returns:

RepeatStatus - indicating whether processing is continuable. Returning null is interpreted as RepeatStatus.FINISHED

Tasklet

- RepeatStatus.CONTINUABLE :

  To inform Spring Batch to run the tasklet again.

  For example, we want to execute a particular tasklet in a loop until a given condition is met, yet we also want to use Spring Batch to keep track of how many times the tasklet was executed.

  Tasklet could return RepeatStatus.CONTINUABLE until the condition is met.

- RepeatStatus.FINISHED.

  We return RepeatStatus.FINISHED, when processing for this tasklet is complete (regardless of success) and to continue with the next piece of processing.

Item Readers & Writers

- All batch processing can be described in its most simple form as
  - reading in large amounts of data,
  - performing some type of calculation or transformation,
  - and writing the result out.
- Spring Batch provides three key interfaces to bulk reading and writing:
  - ItemReader,
  - ItemProcessor,
  - ItemWriter.

Item Reader

- Item Reader : the means for providing data from many different types of input.
  - *Flat-file item readers* : read lines of data from a flat file
  - *XML ItemReaders* : process XML input data & allows for the validation of an XML file against an XSD schema.
  - *Database:* A database resource is accessed to return resultsets which can be mapped to objects for processing.

Item Reader

- ItemReader : An Interface
  - represents the retrieval of input for a Step
  - one item at a time
  - When the ItemReader has exhausted, it indicates this by returning null.

Item Reader

```
public interface ItemReader<T> {
        T read() throws Exception,….
    }
```

- The read method returns one item or null
- The item might represent
  - a line in a file,
  - a row in a database,
  - or an element in an XML file.

  Item is mapped to a usable domain object (such as Trade, Foo, or others),

Item readers types

Item writers types

Item Writer

- ItemWriter : write operations (inverse operations of ItemReader)
- an ItemWriter writes out, rather than reading in.
- In the case of databases or queues, these operations may be inserts, updates, or sends.
- As with ItemReader, ItemWriter is a fairly generic interface

Item Writer

```
public interface ItemWriter<T> {

    void write(List<? extends T> items) throws
                                    Exception;

}
```

Item Writer

- ItemWriter : An Interface
  - represents the output of a Step,
  - one batch or chunk of items at a time.

Item Processor

ItemReader & ItemWriter : useful for their specific tasks

what if we want to insert business logic before writing? Spring Batch provides :

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

Item Processor

- ItemProcessor: An Interface
  - represents the business processing of an item.
  - While the ItemReader reads one item,

    & ItemWriter writes them,
  - *the ItemProcessor provides an access point to transform or apply other business processing*.
  - while processing the item, if the item is not valid, returns null indicating that the item should not be written out.

Item Processor

- An ItemProcessor is simple.
- Given one object, transform it and return another.
- The provided object may or may not be of the same type.
- business logic may be applied within the process,

Item Processor

- An ItemProcessor can be wired directly into a step.
- For example,
- Consider:
  - Read: an ItemReader provides a type Foo
  - *Process: Foo needs to be converted to type Bar*
  - Write : Then type Bar is to be written out

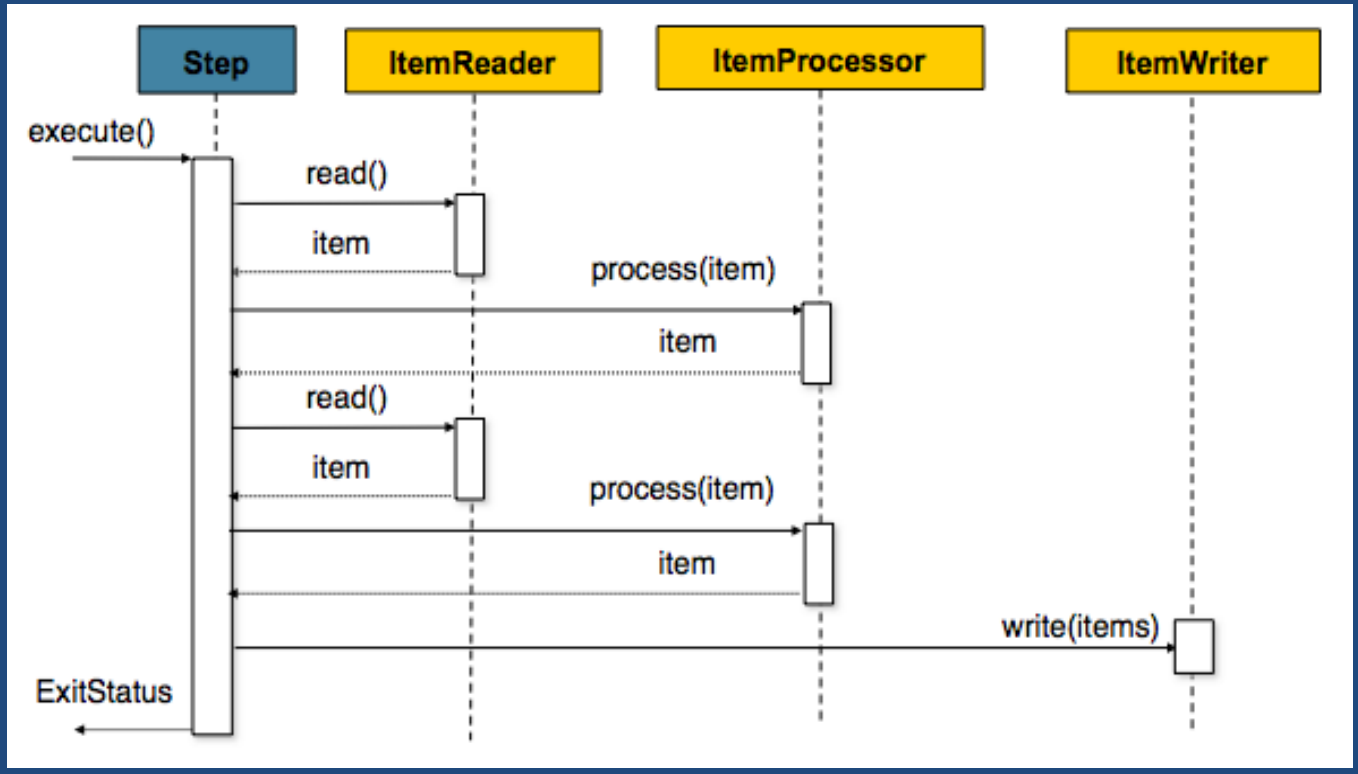
Item Processor

```
public class Foo {}

public class Bar {
    public Bar(Foo foo) {}
}

public class FooProcessor implements ItemProcessor<Foo,Bar>{
    public Bar process(Foo foo) throws Exception {
        //Perform simple transformation, convert a Foo to a Bar
        return new Bar(foo);
    }
}
```

Chunk Oriented Processing

- Chunk oriented processing refers to
  - reading the data one at a time
  - and creating 'chunks'
  - that are written out within a *transaction boundary*.
    - One item is read in from an I*temReader*,
    - handed to an *ItemProcessor*, and *aggregated*.
    - Once the number of items read equals the commit interval, the *entire chunk* is written out by the *ItemWriter*
    - then the transaction is *committed.*

Chunk Oriented Processing



Chunk Oriented Processing

```
List items = new Arraylist();
for(int i = 0; i < commitInterval; i++){
    Object item = itemReader.read()
    Object processedItem =
itemProcessor.process(item);
    items.add(processedItem);
}
itemWriter.write(items);
```

Intercepting Job Execution

- We may want to get some custom code executed during the course of the execution of a Job
- it may be useful to be notified of various events in its lifecycle
- The SimpleJob allows for this by calling a *JobListener* at the appropriate time:

Intercepting Job Execution

```
public interface JobExecutionListener {

    void beforeJob(JobExecution jobExecution);

    void afterJob(JobExecution jobExecution);

}
```

Intercepting Job Execution

annotations corresponding to the interface are:

@BeforeJob

@AfterJob