SPRING ENTERPRISE BATCH

Appreciating All Levels from the Surface to the Depth

Enterprise Batch Processing

- Batch processing is used to process billions of transactions every day for enterprises.
- **EXAMPLE:**

The Options Clearing Corporation[OCC] equity derivatives clearing house

EOD stats: – 12 MM/day - 360 MM per month - 4.2 Billion per year

OCC Batch Processing

- Bulk processing of mission critical operations.
- Time based events (e.g. month-end calculations, notices or correspondence)

periodic application of complex business rules processed repetitively across enormous data sets

(e.g. Insurance benefit determination or rate adjustments)

Integration of information [formatting, validation and processing] in a transactional manner into the system of record.

Modern Java batch applications make use of modern batch frameworks - implementations of JSR 352

Modern Batch Processing

- **GOAL:** Leverage in-house Developer skills for both online and batch processing
  Maximum re-use of implementation.
  Easier development and maintenance, as the same sets of tools are used.
  Consistency in enforcement of enterprise standards and quality of service.

- **Modern Batch Processing :**
  Writing the business logic for the job
  Separation of concern between the business logic and the “plumbing” code
  More efficient modularization of batch functions – cultures re-use

Spring Batch Features

- Transaction management
- Chunk based processing
- Declarative I/O
- Statistics
- Start/Stop/Restart
- Retry/Skip
- Web based administration interface (Spring Batch Admin)
- Parallel processing & partitioning techniques to process high-volume of data.

- Spring Batch

Spring Scheduling Annotation

FixedRate: Interval between method invocations measured from the start time of each invocation.

```
public class MyClass {

    @Scheduled(fixedRate = 5000, initialDelay = 1000)
    public void fixedRateMethod() {
        System.out.println("Fixed rate");
    }

    @Scheduled(fixedDelay = 5000, initialDelay = 2000)
    public void fixedDelayMethod() {
        System.out.println("Fixed delay");
    }

    @Scheduled(cron="0/5 * * * * *")
    public void cronMethod() {
        System.out.println("Cron expression");
    }
}
```

FixedDelay: interval between invocations measured from the completion of the task.

CRON expressions detailed task scheduling.

CRON Scheduling

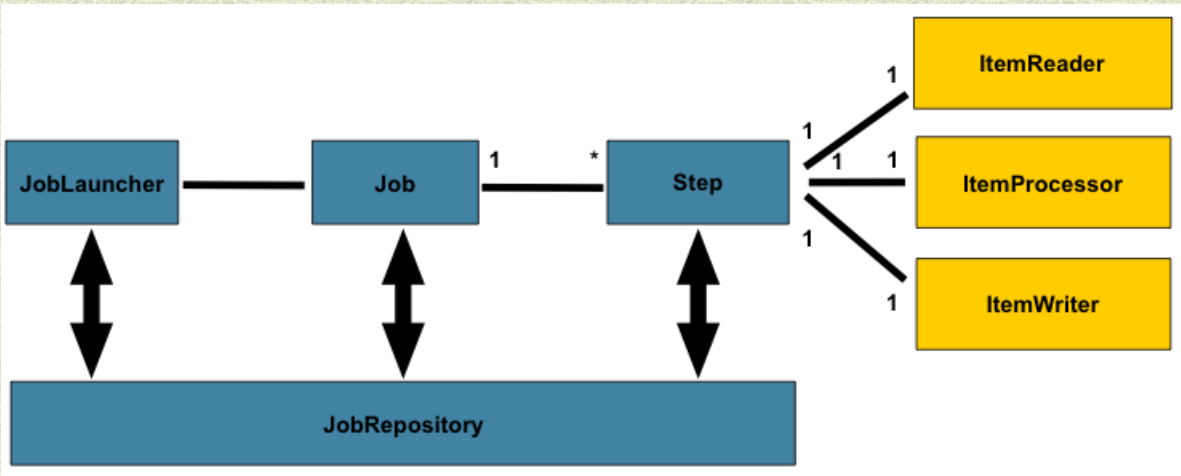
- The pattern is a list of six single space-separated fields:
  second, minute, hour, day, month, weekday. Month and weekday

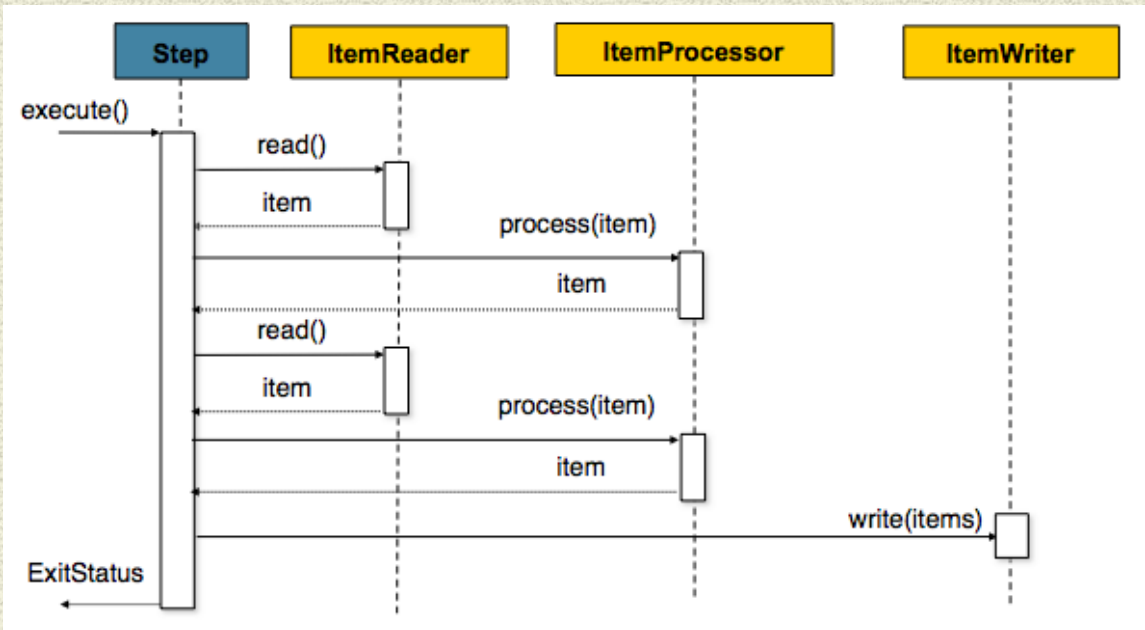
- **Example patterns:**

- "0 0 * * * *" = the top of every hour of every day.
- "*/10 * * * * *" = every ten seconds.
- "0 0 8-10 * * *" = 8, 9 and 10 o'clock of every day.
- "0 0/30 8-10 * * *" = 8:00, 8:30, 9:00, 9:30 and 10 o'clock every day.
- "0 0 9-17 * * MON-FRI" = on the hour nine-to-five weekdays
- "0 0 0 25 12 ?" = every Christmas Day at midnight

CRON: Unix term for time-based job scheduler in Unix computer operating systems
cronExpression – Spring Version

Basic Spring Batch Flow
Extract – Transform – Load





Basic ETL
Extract – Transform – Load
Item Reader, Processor & Writer

Item Reader [Extract]

- **Flat File-** File records with fields of data defined by fixed positions in the file or delimited by some special character (e.g. Comma).
- **XML -** Input data allows for the validation of an XML file against an XSD schema.
- **Database –** Map resultsets to objects for processing.

  The default SQL ItemReaders invoke a RowMapper to return objects, keeps track of the current row if restart is required, store basic statistics…

Item Processor [Transform]

Interface for custom processing

Item Writer [Load]

Inverse operations of ItemReader

Readers-Writers

Job Declaration

- **Declare Reader [Extract] Bean:**
- id="csvFileReader"
  class="org.springframework.batch.item.file.FlatFileItemReader"
- p:resource="classpath:data/products.csv">

- **Declare Mapper [Transform] Bean:**

class="…file.mapping.BeanWrapperFieldSetMapper"
- p:targetType="edu.mum.domain.Product"/>
-
- **Declare Writer[Load] Bean:**
- id="productWriter"
- class="edu.mum.batch.ProductItemWriter">
- <beans:property name="productService">
- <beans:bean class="edu.mum.service.impl.ProductServiceImpl"/>
-

Job Declaration [Cont.]

```
<job job-repository="jobRepository"
        id="SaveProducts">
    <step id="step1">
        <tasklet ref="authenticate"/>
            <next on="*" to="step2" />
    </step>
    <step id="step2">
        <tasklet>
            <chunk commit-interval="5" writer="productWriter"
                                        reader="csvFileReader"/>
        </tasklet>
    </step>
</job>
```

Chunk based processing

1. Start transaction
2. Single item read with ItemReader [*csvFileReader*]
3. Processed by ItemProcessor [*fieldSetMapper*]
4. When # of items read == *commit interval [5]*

   Write chunk with ItemWriter [*productWriter*]
5. Commit transaction

```
<chunk commit-interval="5" writer="productWriter"
                           reader="csvFileReader"/>
```

Error Handling
Skip, Retry, Restart

- **Skip** Logic allows for handling an error that should not cause job failure. The item should be skipped instead. Log bad items for further analysis. [e.g. missing field in record]

- **Retry** attempts an operation several times: the operation can fail at first, but another attempt can succeed.

  Example: DB Lock

- **Restart** the job at point of failure. The work performed by the previous execution isn't lost.

  For example, Skip [or Retry] have exceeded preset limits

Skip Configuration

- Skip up to 2 items [skip-limit], then throw job Failure exception
- Skip for configured exceptions

```
<chunk commit-interval="5" writer="productWriter"
                           reader="csvFileReader" skip-limit="2">
    <skippable-exception-classes>
        <include class="java.lang.Exception"/>
    </skippable-exception-classes>
</chunk>
```

Retry Configuration

- Retry up to 3 times [retry-limit], then throw job Failure exception
- Retry for configured exceptions

```
<chunk commit-interval="5" writer="productWriter"
                           reader="csvFileReader" retry-limit="3“ >
      <retryable-exception-classes>
          <include class="java.lang.Exception"/>
      </retryable-exception-classes>
 </chunk>
```

Restart Execution

```
// Get Failed job instance
JobInstance jobInstance = jobExecution.getJobInstance();

// Restart Job
restartId = jobOperator.restart(jobInstance.getId());

final JobExecution restartExecution =
                        jobExplorer.getJobExecution(restartId);
```

Spring Batch Testing Support

JobLauncherTestUtils

- **End-to-End Testing**
- Test the complete run of a batch job from beginning to end. This allows for a test that sets up a test condition, executes the job, and verifies the end result.

```
jobLauncherTestUtils.launchJob();
```

- **Individual Step Testing**
- For complex jobs, the end-to-end approach is unmanageable
- Allows for targeted tests. Set up data for just that step and to validate its results directly.

```
jobLauncherTestUtils.launchStep("step2ETL");
```

Main Point

Spring Batch provides a systematic, repeatable process for managing the “group dynamics” of large data sets.

The Unified Field and it’s relative correlate the quantum field enhance group dynamics [1% effect]