Spring Batch

Table of Contents

I. Spring Batch Introduction 4

A. Introduction 4

B. Spring Batch Architechture 4

1. Spring Batch xây dựng cấu trúc gồm 3 lớp chính: 4

C. General Batch Principles and Guidelines 7

II. Spring Batch Parallelization 8

A. Multi-threaded Step (single process): 9

B. Parallel Steps 10

C. Remote chunking 11

D. Partitioning: 12

1. PartitionHandler 14

2. Partitioner 15

3. Binding Input Data to Steps 16

III. Domain Language of Batch 17

A. Job 18

1. JobInstance 18

2. JobParameters 19

3. JobExecution 19

B. Step 20

1. Step Execution 20

2. ExecutionContext 21

3. Transactions 21

IV. Configuration 28

A. Main context 28

1. JobRepository 28

2. JobOperator 32

3. JobRegistry 33

B. Job 34

1. Attributes 34

2. Element 35

C. Step 38

1. Attributes 38

2. Element 40

D. Tasklet 43

1. Attributes 43

2. Element 44

E. Chunk 45

1. Attributes 45

2. Element 47

V. Readers, Writers, Processors 49

A. ItemReader 49

B. ItemWriter 50

C. ItemProcessor 50

1. Chaining ItemProcessors 51

2. Filtering Records 52

3. Fault Tolerance 53

D. ItemStream 53

E. The Delegate Pattern and Registering with the Step 53

F. Flat Files 54

1. The FieldSet 54

2. FlatFileItemReader 54

3. LineMapper 55

4. LineTokenizer 55

5. FieldSetMapper 56

6. DefaultLineMapper 56

7. Exception Handling in Flat Files 57

G. FlatFileItemWriter 57

1. LineAggregator 57

2. PassThroughLineAggregator 57

3. FieldExtractor 57

4. PassThroughFieldExtractor 58

5. BeanWrapperFieldExtractor 58

6. Handling File Creation 58

H. XML Item Readers and Writers 59

1. StaxEventItemReader 60

2. StaxEventItemWriter 60

I. Multi-File Input 60

J. Database 61

1. Cursor Based ItemReaders 61

2. Paging ItemReaders 66

3. Database ItemWriters 68

K. Reusing Existing Services 70

L. Validating Input 71

M. Preventing State Persistence 72

N. Creating Custom ItemReaders and ItemWriters 72

1. Custom ItemReader 72

2. Custom ItemWriter 74

VI. Parallelization configuration 74

A. Multi-threaded: 74

B. Parallel step: 75

C. Remote chunking: 76

1. Config master 76

2. Config slave: 77

D. Partitioning: 78

1. Partitioning multi thread: 78

2. Partitioning remote node: 79

3. Config slave 81

VII. Repeat and Retry 82

A. Repeat: 82

1. Repeat template: 82

2. Completion Policies: 84

3. Exception handling: 84

4. Listerners: 85

5. Parallel processing: 85

6. Declarative iteration: 86

B. Retry: 87

1. RetryTemplate 87

2. Stateless Retry 89

3. Stateful Retry 89

4. Retry Policies 90

5. Backoff policies 92

6. Listeners 92

7. Declarative Retry 93

VIII. Logging 94

A. Logging Item Processing and Failures 94

B. Stopping Job Manually for Business Reasons 95

C. Adding a Footer Record 96

1. Writing a Summary Footer 97

D. Driving Query Based ItemReaders 99

E. Multi-Line Records 100

F. Executing System Commands 102

G. Handling Step Completion When No Input is Found 102

H. Passing Data to Future Steps 103

IX. Unit Testing 104

A. End-To-End Testing of Batch Jobs: 105

B. Testing Individual Steps: 106

C. Testing Step-Scoped Components: 106

D. Validating Output files: 108

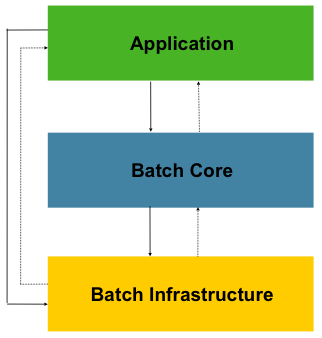
E. Mocking Domain Objects: 108

# Spring Batch Introduction

## Introduction

* Batch Processing là quá trình thực hiện hàng loạt các công việc mà không cần sự can thiệp của con người.
* Spring Batch là framework mã nguồn mở sử dụng batch processing.
* Spring Batch xây dựng những chức năng có thể tái sử dụng trong việc xử lý dữ liệu lớn, bao gồm:
* Theo dõi quá trình hoạt động của batch process.
* Quản lý truy xuất dữ liệu với database.
* Thống kê quá trình xử lý công việc.
* Quản lý trạng thái công việc: start, stop,restart
* Quản lý code.

## Spring Batch Architechture



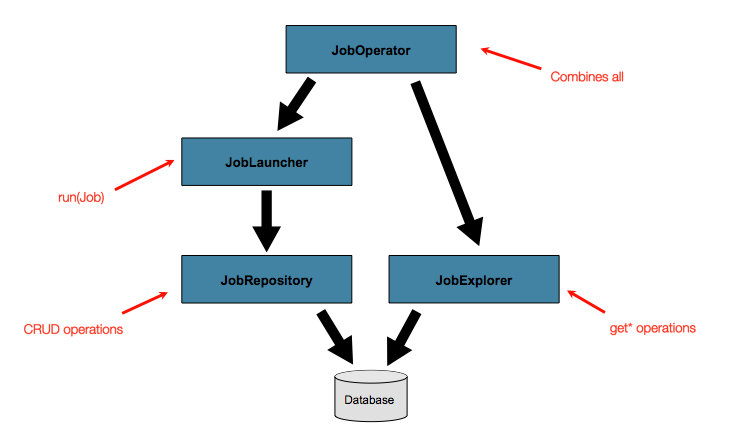
### Spring Batch xây dựng cấu trúc gồm 3 lớp chính:

#### Application layer

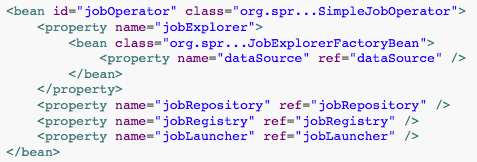
* Batch job là tập hợp hàng loạt những công việc thực hiện được thiết lập sẵn trong file
* Application layer chứa tất cả các batch job và code của người viết.

#### Core Layer

* Core layer chứa các classes cần thiết để thực thi công việc và kiểm soát quá trình hoạt động của công việc đó bao gồm:
* JobOperator
* JobLauncher
* JobExplorer
* JobRepository



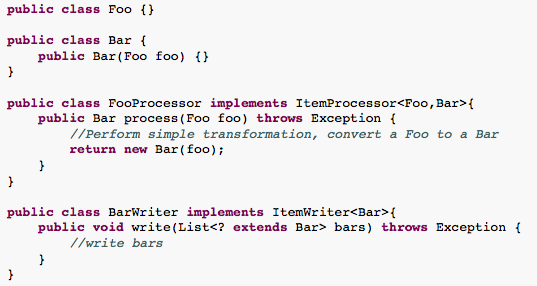
* JobOperator
  + Dùng để quản lý quá trình thực thi công việc như: start, stop,restart.
  + Để truy xuất thì JobOperator gọi JobExplorer
* JobLauncher
  + nhận yêu cầu từ JobOperator ,sau đó gọi JobRepository tạo instance của job và run job
* JobExplorer
  + cung cấp quyền truy cập thông số của job trong JobRepository.
* JobRepository
  + cung cấp những chức năng như : create,update,delete để thực hiện yêu cầu của JobLauncher , thực thi job, steps.



#### Infrastructure Layer

* Infrastructure layer chứa reader and writer và service như là RetryTemplate.
* Infrastructure sử dụng những thư viện có sẵn trong Spring Batch Framework :
* ItemReader
* ItemWriter
* ItemProcessor

* ItemReader
  + cung cấp dữ liệu từ 3 kiểu dữ liệu chính là:
    - Flat - File
    - Xml - File
    - Database
* ItemProcessor
  + xử lý dữ liệu trung gian từ ItemReader xuất ra ItemWriter.





* ItemWriter
  + lưu trữ dữ liệu từ ItemProcessor thành 3 loại chính giống như item reader

## General Batch Principles and Guidelines

- Tránh xây dựng logic phức tạp cho việc xử lý hàng loạt công việc.

- Nên lưu những dữ liệu cần thiết ở cùng 1 nơi.

- Hạn chế sử dụng truy xuất dữ liệu với database,lâu hơn truy xuất trên bộ nhớ(In Memory)

- 1 công việc k xử lý 1 dữ liệu 2 lần khi đã thực thi xong.

- Đảm bảo dữ liệu khi được đoc, xử lí, viết được đầy đủ.

- Đảm bảo bộ nhớ khi công việc được thực thi.

- Kiểm tra dữ liệu 1 cách chặt chẽ

- Nên:

* Đọc dữ liệu 1 lần và lưu trữ để những xử lý khác có thể tái sử dụng
* Câu lệnh sql hạn chế dùng where với index

# Spring Batch Parallelization

- Có hai cách tiếp cận để mở rộng quy mô hệ thống để tăng hiệu suất:

* **Vertical Scaling** – Faster machine with more RAM, CPUs, GHz
* **Horizontal Scaling** – Add more machines to distribute load remotely

- Ở mức độ cao nhất thì “parallel processing” có 2 modules: single process, multi-thread và multi-proces.

Chúng được chia thành các loại như sau:

* **Parallel Steps** (single process) (Vertical Scaling)
* **Multi-threaded Step** (single process) (Vertical Scaling)
* **Remote Chunking of Step** (multi process) (Horizontal Scaling)
* **Partitioning a Step** (single or multi process) (Horizontal Scaling)

## Multi-threaded Step (single process):

- Các đơn giản nhất để start “parallel processing” là thêm “TaskExecutor” vào file configuration, ex: attribute của tasklet:

<step id="loading">

<tasklet task-executor="taskExecutor">...</tasklet>

</step>

- Trong ví dụ trên, taskExecutor liên hệ đến một bean khác, triển khai TastExecutor interface. TaskExecutor là interface tiêu chuẩn của Spring, vì vậy xem thêm phần Spring User guide để biết cụ thể qá trình triển khai. Một multi-thread đơn giản về tastExecutor là SimpleAsyncTaskExecutor.

- Kết quả của cấu hình bên sẽ dẫn đến: quá trình Step chạy bởi read, process và write mỗi chunk of item là trong từng thread riêng lẽ của quá trình chạy. Chú ý là điều có nghĩa là không có thứ tự cố định cho các mặc hang để được xử lý và một đoạn có thể chưá item không liên tục so sánh vs singe-thread. Thêm nữa là, bất kì vấn đề limit không gian bởi task executor, thì sử dụng throttle limit trong task config vs mặc định là 4 và tất nhiên bạn có thể thay thế số lượng đó:

Ex:

<step id="loading">

<tasklet task-executor="taskExecutor" throttle-limit="20">...</tasklet>

</step>

- Cũng lưu ý rằng, có thể có limits places được đặt đồng thời các datasource. Hãy chắc chắn rằng các pool trong resources ở mức thấp đủ lớn để đáp ứng đủ ố lượng thread cũng chạy đồng thời tong step.

- Practical limitations:

+ Many participants trong một step (read, write), và nếu chúng không được tách biệt bởi thread thì chúng sẽ trở nên không có tác dụng trong multi-thread step.

+ Đặc biệt nhất là trong “the off-the-shelf” đọc và ghi trong Spring batch không được thiết kế cho multi-process sử dung.

- Spring batch cung cấp các phương thức: ItemWriter và ItemReader. Bạn có thể kiểm tra các lớp này có phải là threadsafe hay không trong javadoc hoặc các nơi lưu thong tin lien quan. Nếu reader không phải là thread safe, nó có thể ảnh hưởng nó có thể sử dụng hiệu quả trong quá trình đồng bộ riêng của bạn trong delegator. Bạn có thể đồng bộ gọi phương thức read và miễn là nó đang trong processing và writing là một phần tốn nhiều tài nguyên của chunk, step của bạn có thể hoàn thành nhanh hơn vs kiểu cấu hình single threaded.

## Parallel Steps

- Trong những ứng dụng logic mà nó cần phải parallized có thể chia ra nhiều phần trách nhiệm riêng biệt và assigned tới từng step riêng thì sau đó nó có thể parallized in a singe process. Paralled step execution được cấu hình một cách dễ dàng, ví dụ: triên khai step(step1, step 2) trong parallel vs step3, bạn có thể cấu hình :

<job id="job1">

<split id="split1" task-executor="taskExecutor" next="step4">

<flow>

<step id="step1" parent="s1" next="step2"/>

<step id="step2" parent="s2"/>

</flow>

<flow>

<step id="step3" parent="s3"/>

</flow>

</split>

<step id="step4" parent="s4"/>

</job>

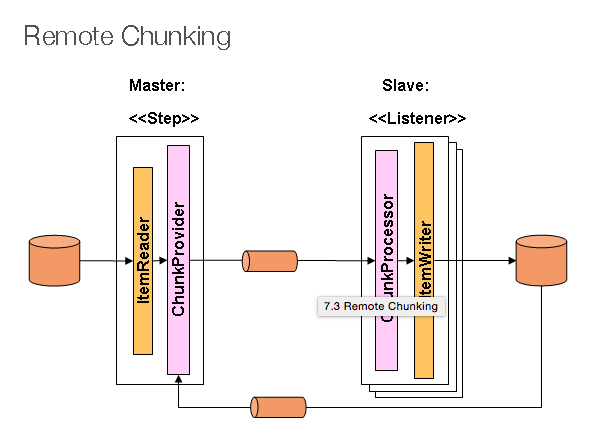
<beans:bean id="taskExecutor" class="org.spr...SimpleAsyncTaskExecutor"/>

- Cấu hình thành phần ”task-executor” được sử dụng cụ thể vs triển khai TaskExecutor nên được sử dụng chạy trong luồng cá nhân. Mặc định là SyncTaskExecutor, nhưng kiểu không đồng bộ TaskExecutor được yêu cầu chạy step trong parallel.

- Chú ý là JOB được bảo đảm là mỗi flow được chia ra trước khi tập hợp các output và chuyển.

## Remote chunking

- Trong remote chunking, Step processing được chia qua nhiều processes, giao tiếp vs nhau thông qua middleware.



- Phần Master là một single process và slave là multi remote process. Rõ ràng mô hình này hoạt động tốt nếu Master không có nút thắc cổ chai, vì vậy quá trình phải tốn nhiều tài nguyên hơn trong

reading item.

- Master chỉ là phần triển khai của Spring batch step, vs ItemWriter được thay thế với các generic version mà được biết là chuck of items sẽ được gởi đi như thế nào đến middleware dạng message.

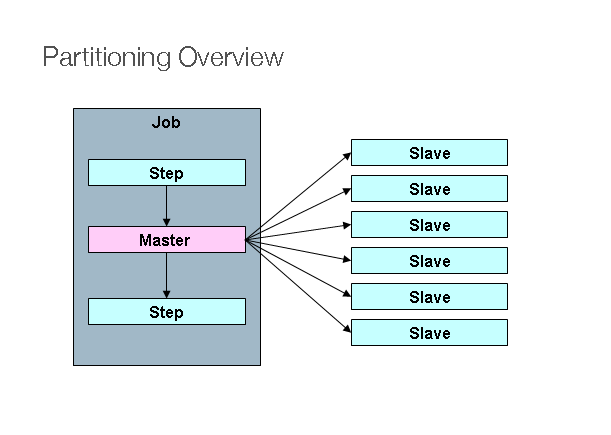
- Slaves là một dạng lắng nghe tiêu chuẩn cho bất kì yêu cầu gì từ middleware đang được sử dụng, và mục đích của chúng là process chunk of items sử dụng chuẩn ItemWriter or ItemProcessor plus ItemWriter thông qua interface ChunkProcessor.

Adv: reader, processor and writer components là off-the-shelf (giống vs việc được dùng cho các step được thực hiện ở local).

Item được phân chia tự động và làm việc được chia thông qua middleware, vì vậy nêú listeners là những bộ phận năng động, thì load balancing là tự động.

## Partitioning:

- Spring batch cung cấp SPI cho partitioning Step execution.

- Remote participants là một step đơn giản mà nó co thể dễ dàng cấu hình và dùng nó trong quá trình chạy ở local.

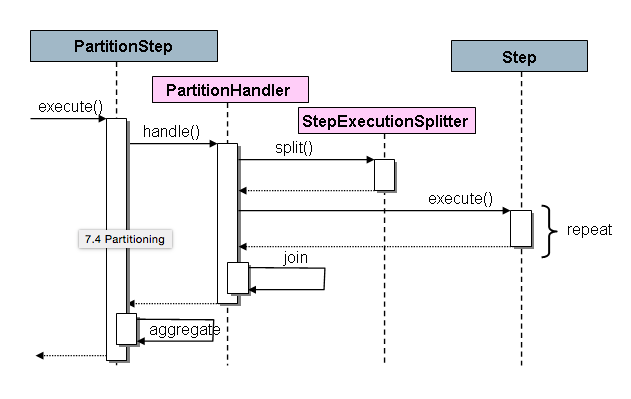
- Các JOB được thực hiện ở phía bên tay trái là một chuỗi STEP, và một trong những STEP này được gọi là Master.

- Các Slaves trong ảnh này là tất cả các trường hợp y hệt một STEP, STEP trong thực tế có thể chiếm chỗ của các MASTER để dẫn đến cùng một kết quả cho JOB.

- Các Slave thường là remote services, nhưng cũng có thể là local threads.

- Các tin nhắn được gửi bởi MASTER đến Slave trong mô hình này không cần phải có độ bền, hoặc đã được đảm bảo deliver : Spring batch metadata trong các JobRepository sẽ đảm bảo rằng mỗi Slave được thực hiện một lần và chỉ một lần cho mỗi lần thực thi 1 công việc.

- SPI trong Spring batch bao gồm việc thực hiện đặc biệt của STEP (PartitionStep), và hai giao diện chiến lược đó cần phải được thực hiện trong môi trường cụ thể. Các interface chính là PartitionHandler và StepExecutionSplitter, và vai trò của họ là hiển thị trong sơ đồ trình tự dưới đây:



- Step bên phải trong trường hợp này là "remote" Slave, vì vậy có khả năng có nhiều đối tượng và các quá trình chạy trên đó, và PartitionStep được thực hiện execution. Các cấu hình PartitionStep như sau:

<step id="step1.master">

<partition step="step1" partitioner="partitioner">

<handler grid-size="10" task-executor="taskExecutor"/>

</partition>

</step>

- Tương tự như multi-threaded step's throttle-limit attribute, các grep-size quy định số lượng thread.

### PartitionHandler

- Các PartitionHandler là thành phần hiểu về fabric của remoting hoặc grid environment. Nó có thể gửi yêu cầu StepExecution để điều khiển STEP, bao bọc một số định dạng fabric-specific..

- Nó không phải biết làm thế nào để phân chia các dữ liệu đầu vào, hoặc làm thế nào để tổng hợp kết quả của multiple Step.

- Nói chung thì nó cũng không cần phải biết về khả năng phục hồi hoặc chuyển đổi dự phòng, bởi vì đó là tính năng của fabric trong nhiều trường hợp, và Spring luôn luôn cung cấp hàng loạt restartability độc lập của fabric: một failed JOB luôn luôn có thể được khởi động lại và chỉ STEP thất bại sẽ được tái thực hiện.

- Interface PartitionHandler có thể có việc thực hiện triển khai cụ thể cho một loạt các loại faric: ví dụ: đơn giản RMI remoting, EJB từ xa, dịch vụ web tùy chỉnh, JMS, Spaces Java, chia sẻ memory grids (như Terracotta hoặc Coherence), grip faric (như GridGain).

- Spring batch không cung cấp một thực hữu ích của PartitionHandler mà thực hiện Step tại từng local riêng biệt trong khâu thực hiện, sử dụng các chiến lược TaskExecutor từ Spring. Việc thực hiện được gọi là TaskExecutorPartitionHandler, và nó là mặc định cho một bước cấu hình với các không gian tên XML như trên. Nó cũng có thể được cấu hình một cách rõ ràng như thế này:

<step id="step1.master">

<partition step="step1" handler="handler"/>

</step>

<bean class="org.spr...TaskExecutorPartitionHandler">

<property name="taskExecutor" ref="taskExecutor"/>

<property name="step" ref="step1" />

<property name="gridSize" value="10" />

</bean>

- Các gridSize xác định số lượng các STEP khác được tạo ra, vì vậy nó có thể được kết hợp với kích thước của thread pool trong TaskExecutor, hoặc nếu không nó có thể được thiết lập để được lớn hơn số lượng các chủ đề có sẵn.

- Các TaskExecutorPartitionHandler là khá hữu ích cho IO bước sau, giống như sao chép số lượng lớn các tập tin hoặc sao chép hệ thống tập tin vào hệ thống quản lý nội dung. Nó cũng có thể được sử dụng để thực hiện từ xa bằng cách cung cấp một thực hiện STEP đó là một proxy cho một lời gọi từ xa (ví dụ như sử dụng Spring Remoting).

### Partitioner

- Các Partitioner có trách nhiệm đơn giản: để tạo ra nội dung execution theo các thông số đầu vào cho bước hành mới (không cần khởi động lại). Nó có một method:

**public** **interface** Partitioner {

Map<String, ExecutionContext> partition(**int** gridSize);

}

- Sự trả về giá trị từ method liên kết vs một tên unique của mỗi STEP execution., vơi tham sô truyền vào trong form của ExecutionContext

- Cái tên đuọc show sau cùng trong Batch metadata giống như tên STEP name trong partition StepExecution.

- ExecutionContext nó chỉ là một gói trong một cặp giá trị, vì vậy nó chứa range of primary keys, line number, địa điểm của input file.

- SimplePartitioner: tên của các Step execution(key được return trong Map return bởi partitioner) cần phải được unique giữa các STEP được execution trong JOB. Một các đơn giản để làm việc này và đặt tên dễ hiểu thì có một quy ước là prefix + suffix, prefix: tên của STEP mà nó chạy, suffix chỉ là một counter.

- Tuỳ chọn PartitioneNameProvider có thể cung cấp tên partition riêng lẽ từ partition của chính nó. Nếu một partitione triển khai interface rồi khi restart chỉ có tên được truy vấn.

### Binding Input Data to Steps

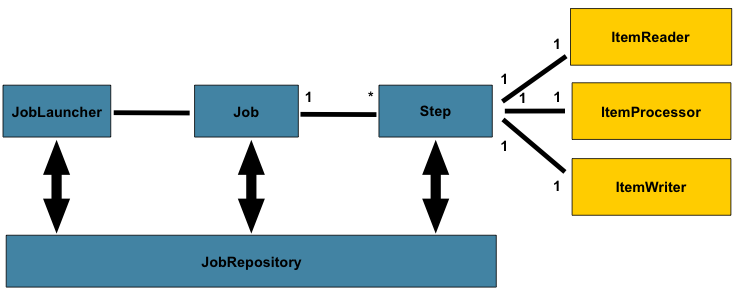
- Nó rất hiệu quả cho các Step được thực hiện bởi các PartitionHandler có cấu hình giống hệt nhau, và cho các thông số đầu vào của họ bị ràng buộc trong thời gian chạy từ ExecutionContext. Điều này dễ dàng để làm với các tính năng StepScope của Spring Batch.

- Ví dụ, nếu các phân vùng tạo ra các trường hợp ExecutionContext với filename attribute, chỉ vào một tập tin khác nhau (hoặc thư mục) cho mỗi lần gọi STEP, Partitioner có thể trông như thế này:

**Example step execution name to execution context provided by Partitioner targeting directory processing**

|  |  |
| --- | --- |
| **Step Execution Name (key)** | **ExecutionContext (value)** |
| filecopy:partition0 | fileName=/home/data/one |
| filecopy:partition1 | fileName=/home/data/two |
| filecopy:partition2 | fileName=/home/data/three |

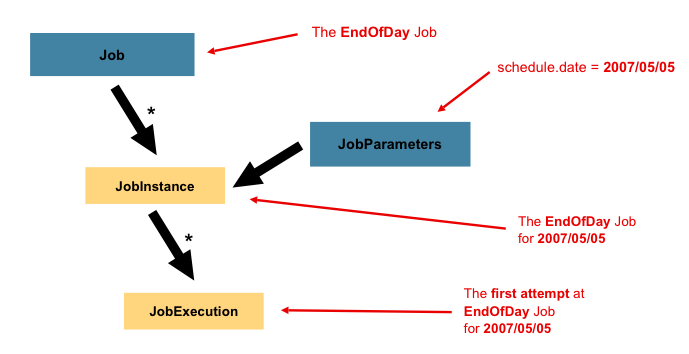
# Domain Language of Batch



Những thành phần chính tạo nên Spring Batch App là:

* JobRepository
* JobLauncher
* Job
* Step
* Tasklet
* Item
* Chunk
* ItemReader
* ItemProcessor
* ItemWriter

## Job



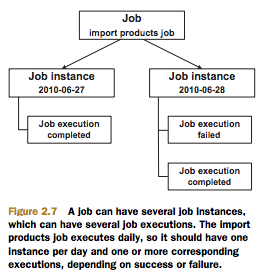
### JobInstance

* Bao gồm:
  + Job
  + JobParameters
* Ex:
  + Figure 2.7: Job ở đây là ( import Products ) , Job này có 2 JobInstance, 2 JobInstance này được phân biệt bởi JobParameters(2010-06-27 và 2010-06-28).JobInstance có thể có 1 hoặc nhiều JobExecution,JobInstance chỉ chạy thành công khi JobExecution trả về completed.
  + Code:

jobLauncher.run(job,new JobParametersBuilder()

.addString(“date”,”2010-06-27”)

.toJobParameters());

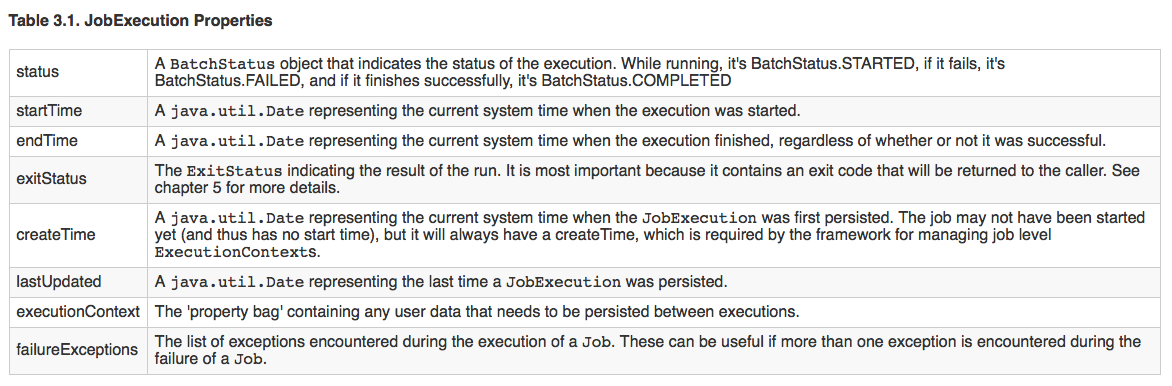
* 

### JobParameters

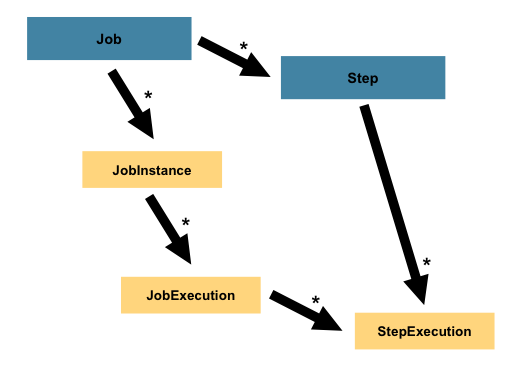
* Phân biệt các JobInstance với nhau, thường mang gía trị thời điểm khởi tạo

### JobExecution

* JobInstance chỉ chạy thành công khi JobExecution tương ứng trả về success(JobExecution có thể trả về success hoặc failure)
* Ex:
  + Lần đầu tiên chạy JobInstance (có JobParameters là: 01-01-2008) JobExecution trả về trạng thái failure.Nếu JobInstance đó chạy lại với JobParameters là 01-01-2008 thì JobExecution sẽ được tạo mới,và nó có thể trả về success.Tuy nhiên chỉ có 1 JobInstance được chạy.(1 JobInstance có thể có 1 hoặc nhiều JobExecution,tùy thuộc vào trạng thái JobExecution trả về)
* JobExecution sẽ chứa những trạng thái, những thông số liên quan tới quá trình thực thi 1 JobInstance.

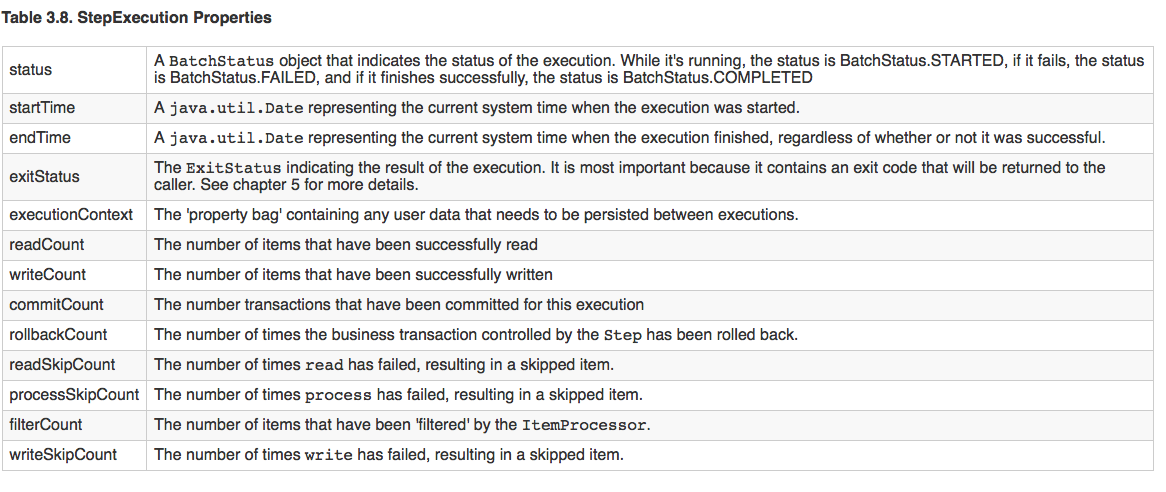


## Step



### Step Execution

* Giống JobExecution,StepExecution chỉ được khởi tạo khi thực thi Step, tuy nhiên 1 Step được thực thì mà Step trước đó trả về fails thì Step này sẽ không khởi tạo StepExecution được, vì vậy mà nó sẽ không có StepExecution.Có nghĩa là : StepExecution chỉ được khởi tọa khi Step trước Step đang thực thi trả về success.
* Mỗi StepExecution sẽ chứa những trạng thái và thông tin liên quan tương ứng với Step và JobExecution và những dữ liệu liên quan như là commit và rollback count và start , endtimes.
* Mỗi StepExecution khi thực thi sẽ chứa ExecutionContext,vì vậy nó sẽ chia sẻ dữ liệu cần thiết cho việc chạy spring batch, nó như là thống kê hoặc trạng thái, thông tin cần thiết cho việc restart quá trình thực thi của spring batch.



### ExecutionContext

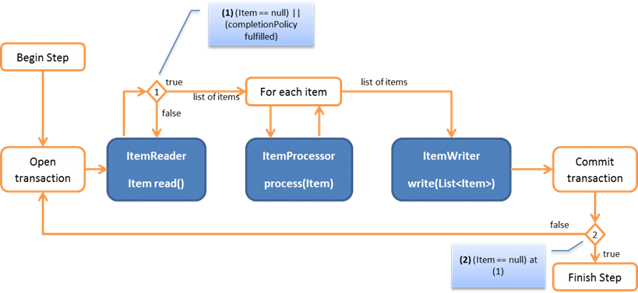
* ExecutionContext như là nơi lưu trữ giá trị các biến giữa trong StepExecution hoặc JobExecution.
* ExecutionContext chỉ tồn tại trong khoảng thời gian StepInstance hoặc JobInstance được khởi tạo.

### Transactions

Spring Batch xử lý các transaction tại các step. Spring Batch sẽ không bao giờ sử dụng duy nhất một transaction cho toàn bộ job – trừ khi job đó chỉ có một step.

Có 2 cách thực hiện job trong Spring Batch: dùng tasklet hoặc dùng chunk-oriented steps.(tạm dịch: các bước “khối định hướng”)

#### Chunk-oriented steps with transaction



Hình 4.1 - Chunk-oriented with transaction

- Trong chunk-oriented processing, chúng ta có:

· ***ItemReaders***: đọc items, lần lượt từng item, luôn cung cấp item tiếp theo. Khi không còn item nào, Reader sẽ trả về *null*.

· ***ItemProcessors***: (tùy chọn) lấy một item và cung cấp một item.

· ***ItemWriters***: lấy ra một danh sách items và ghi chúng vào một nơi nào đó(database, xml…).

- The batch được tách ra thành nhiều chunks(khối) và mỗi chunk chạy transaction của riêng nó.

- Kích thước của các chunks được xác định bằng ***CompletionPolicy***. Mặc định khi chúng ta dùng *commit-interval=”x”* (còn gọi là *SimpleCompletionPolicy*) trong chunk, kích thước của chunk tương ứng với x(số lượng item). Hoặc chúng ta có thể tạo riêng ***CompletionPolicy*** với đặc tính(attribute) *chunk-completion-policy*.

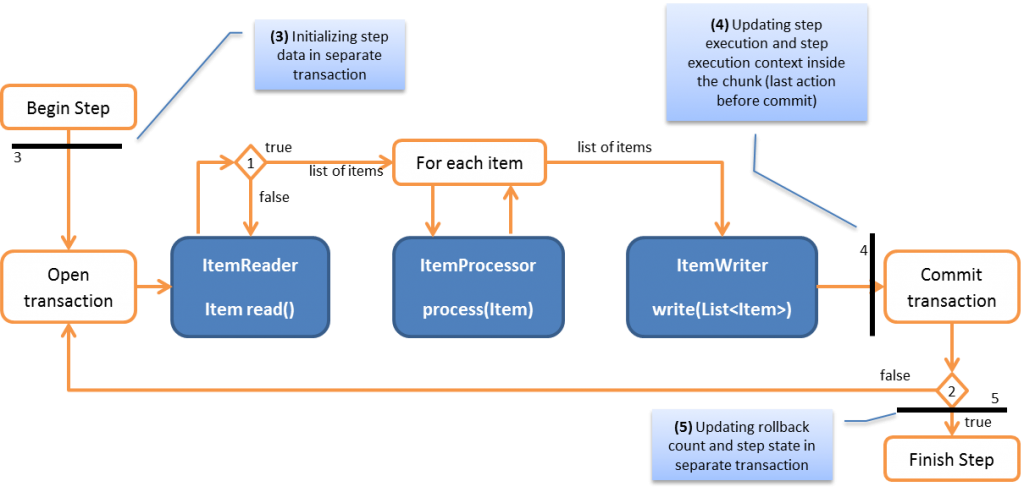
VD: <batch:chunk reader=”myItemReader” writer=”myItemWriter” commit-interval=”10” />

- Nếu xảy ra ngoại lệ (RuntimeException,…), transaction của chunk sẽ bị roll back và the batch thất bại. Các chunk đã được committed vẫn được giữ trong trạng thái đã xử lý(processed state).

#### Business data and batch job data

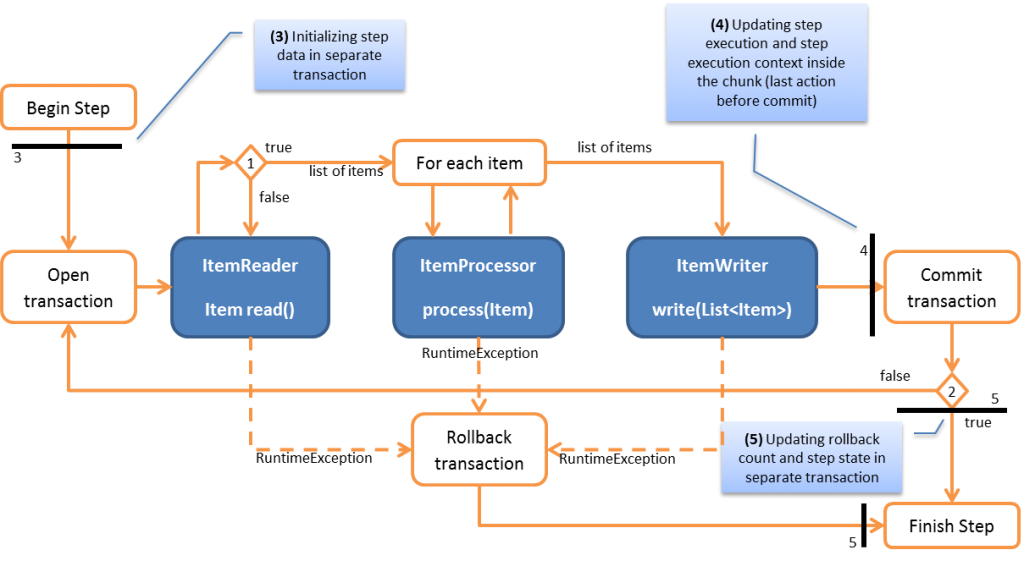
Spring Batch có một bộ bảng dữ liệu được định nghĩa sẵn, được dùng để lưu dữ liệu về job, steps, những thay đổi của job, step.

Lời khuyên: không nên đặt những bảng này trong một database khác ngoài business data. Job data bao gồm trạng thái của job, steps, processed items, start time, end time, state (COMPLETED, FAILED, …) …Những thay đổi trong dữ liệu này phải nằm trong transaction chạy trên business data. Nếu chúng ta có hai database, chúng ta sẽ cần dùng JtaTransactionManager để xử lý DataSources khác nhau.



Hình 4.2 - Chunk-oriented with successful steps

#### A failed batch



Hình 4.3 - A failed batch

- Khi xảy ra một ngoại lệ ở đâu đó (RuntimeException), transaction sẽ bị roll back, step bị đánh dấu FAILED, và job sẽ thất bại. Bước (5) chắc chắn rằng trạng thái thất bại lưu vào database.

- Chúng ta có thể lựa chọn khi xảy ra ngoại lệ thì transaction không bị roll back bằng cách dùng *no-rollback-exceptions*.

VD: <batch:tasklet>

<batch:chunk…/>

<batch:no-rollback-exception-classes>

<batch:include class=”com.mulodo.MyRuntimeException”/>

<batch:no-rollback-exception-classes>

</batch:tasklet>

#### Các đặc tính của Transaction (Transaction attributes)

- Đặc tính gồm: *isolation, propagation, timeout*.

VD: <batch:tasklet>

<batch:transaction-attributes isolation=”READ\_COMMITTED” propagation=”REQUIRES\_NEW” timeout=”200”/>

<batch:chunk reader=”myItemReader” writer=”myItemWriter” commit-interval=”20”/>

</batch:tasklet>

- Nếu không cấu hình thì String Batch sẽ chạy mặc định: isolation=”DEFAULT”, propagation=”REQUIRED”.

- Tham khảo isolation khác [tại đây](http://en.wikipedia.org/wiki/Isolation_%28database_systems%29).

#### Cursor based reading

- Đọc dữ liệu từ cursor(con trỏ) trong database có nghĩa ta phải mở một kết nối, chạy câu lệnh SQL, đọc từng dòng trong suốt cả job. Chỉ có một vấn đề: khi committing transaction sẽ đóng kết nối. Để giữ kết nối mở, ta dùng JdbcCursorItemReader.

- Trong môi trường application server, thông thường chúng ta lấy kết nối từ một DataSource quản lý bởi application server, và tất cả các kết nối sẽ thông qua transaction theo mặc định. Chúng ta cần phải tạo một DataSource riêng biệt không thông qua transaction, và **chỉ** tiêm(inject) nó vào cursor based readers.

#### Restarting a failed batch

- Một Job instance được xác định bởi *JobParameters*, do đó khi execution xảy ra failed lần đầu tiên, thì job đã start với các parameters – đã dùng trong job execution trước – tự động restart. Nếu không thì job execution tiếp theo sẽ bị từ chối.

- Vậy chúng ta chỉ restart các failed job? Tất nhiên là không. Người nào đó phải biết nơi gọi lại failed job lần nữa. Lớp con của Readers: *AbstractItemCountingItemStreamItemReader* lưu trữ số lượng item vào *ExecutionContext* – được tạo trong mỗi chunk transaction .

Vd: Chúng ta có chunk size là 5 và đang xảy ra lỗi tại item 23. Lần cuối cùng có transaction committed thành công chứa các item 16 đến 20, cho nên số item lưu trong *ExecutionContext* trong database là 20. Khi đang restart job, chúng ta sẽ tiếp tục với item 21 ( và hy vọng là đã sửa lỗi gây ra problem với item 23).

- Hoặc có thể dùng lớp con: *JdbcCursorItemReader*. Chúng ta tạo SQL query, và muốn gọi hàm restart, vậy thì chúng ta phải chắc rằng SQL query sẽ trả về kết quả như khi gọi restart, ít nhất là tất cả các item đã được processed.

Vd: Khi restart với item số 21 thì chúng ta phải chắc chắn rằng item từ 1 đến 20 đã được processed trong lần thử đầu tiên, nếu không thì chúng ta sẽ không lấy được kết quả trả về như mong muốn.

#### Listeners and transactions

- Một số listener bên cạnh *ItemReaders, ItemProcessors, ItemWriters* như:

· *JobExecutionListener*: có 2 method *beforeJob, afterJob*. Được thực thi (executed) bên ngoài chunk’s transaction.

· *StepExecutionListener*: có 2 method, *beforeStep, afterStep*. Được thực thi bên ngoài chunk’s transaction.

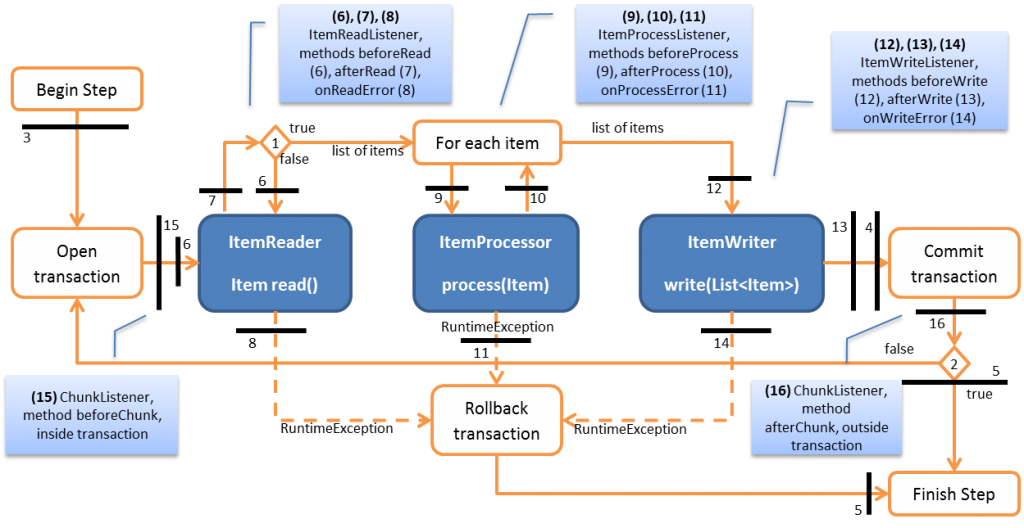
· *ChunkListener*: có 2 method, *beforeChunk-thực thi bên trong chunk’s transaction, afterChunk*-thực thi bên ngoài chunk’s transaction.

· *ItemReadListener*: có 3 method, *beforeRead, afterRead, onReadError*. Được thực thi bên trong chunk’s transaction.

· *ItemProcessListener*: có 3 method *beforeProcess, afterProcess, onProcessError*. Được thực thi bên trong chunk’s transaction.

· *ItemWriteListener*: có 3 method *beforeWrite, afterWrite, onWriteError*. Được thực thi bên trong chunk’s transaction.

· *SkipListener*: có 3 method, onSkipInRead, onSkipInProcess, onSkipInWrite. Được thực thi bên trong chunk’s transaction.



Hình 4.4 - Listeners

- Từ hình trên, chúng ta cần chú ý một điều quan trọng là các method onXXXError được thực thi ngay trước khi transaction rollback. Do dó, nếu chúng ta muốn tương tác với một vài loại transactional resource thì chúng ta cần phải mở một transaction mới bằng annotation ví dụ như: @Transactional(propagation = Propagation.REQUIRES\_NEW) trên hàm muốn dùng.

#### Skip

Có 2 cách khai báo skip trong Spring Batch: dùng *skip-limit* trên chunk và dùng *skippable-exception-classes* bên trong chunk.

VD: <batch:tasklet>

<batch:chunk reader=”myItemReader” writer=”myItemWriter” commit-interval=”20” skip-limit=”15”>

<batch:skippable-exception-classes>

<batch:include class=”com.mulodo.MySkippableException”/>

<batch: skippable-exception-classes >

</batch:chunk>

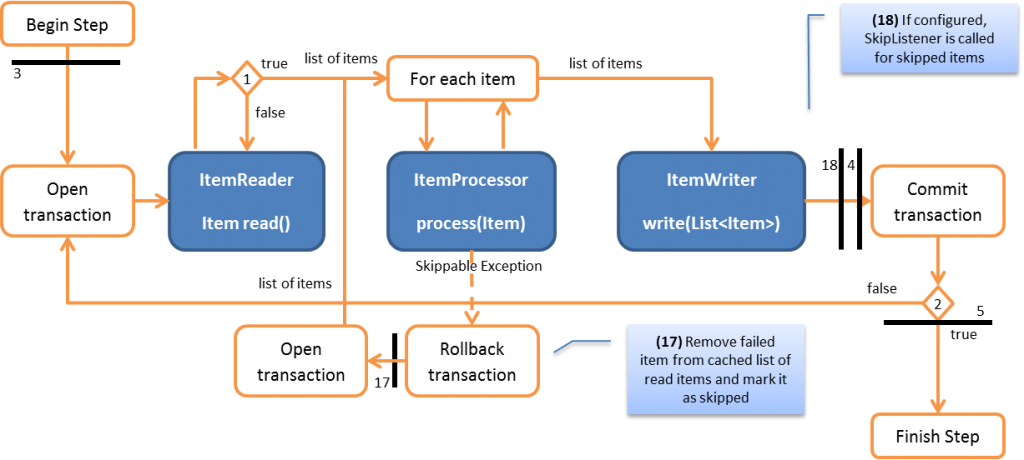
</batch:tasklet>

- Có thể tự tạo các qui định để skip theo ý muốn. Dùng SkipPolicy interface trong chunk, *skip-limit* và *skippable-exception-classes* bị bỏ qua.

VD: <batch:tasklet>

<batch:chunk reader=”myItemReader” writer=”myItemWriter” commit-interval=”20” skip-policy=”mySkipPolicy”/>

</batch:tasklet>



Hình 4.5 - Skip

#### Retry

- Có 2 cách khai báo retry trong Spring Batch: dùng *retry-limit* trên chunk và *retryable-exception-classes* bên trong chunk.

VD: <batch:tasklet>

<batch:chunk reader=”myItemReader” writer=”myItemWriter” commit-interval=”20” retry-limit=”15”>

<batch:retryable-exception-classes>

<batch:include class=”com.mulodo.MyRetryableException”/>

<batch: retryable-exception-classes >

</batch:chunk>

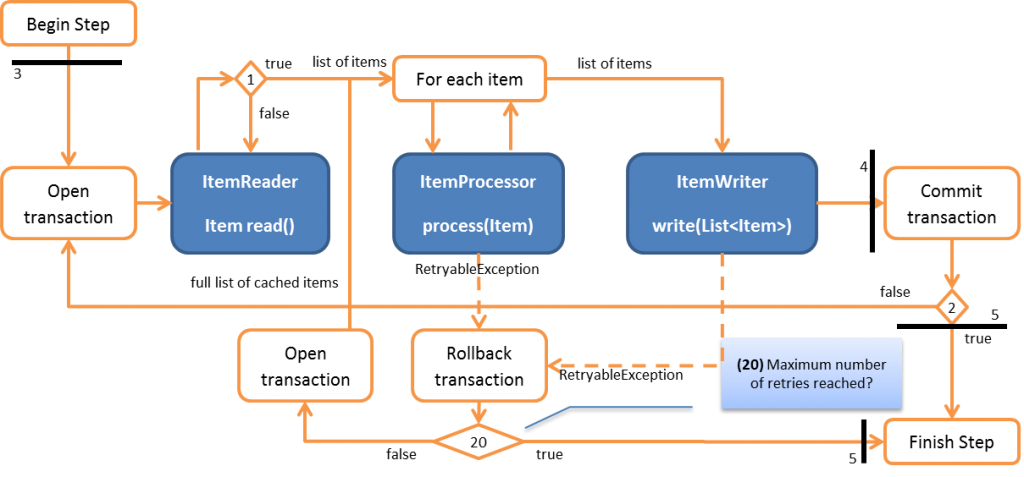
</batch:tasklet>

- Có thể tự tạo các qui định để retry theo ý muốn. Dùng RetryPolicy interface trong chunk.

VD: <batch:tasklet>

<batch:chunk reader=”myItemReader” writer=”myItemWriter” commit-interval=”20” retry-policy=”mySkipPolicy”/>

</batch:tasklet>



Hình 4.6 - Retry

# Configuration

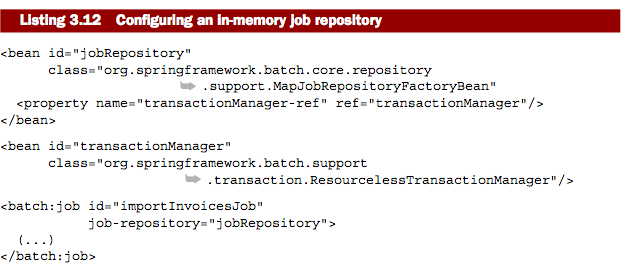
## Main context

### JobRepository

* Spring Batch cung cấp class SimpleJobRepository(set DataSet) sử dụng cho mục đích quản lý riêng.
* SimpleJobRepository gồm 2 loại:
* In-Memory with no persistent
* Persistent with metadata use JDBC
* In-Memory DAO có thể dùng để test, nhưng bạn nên sử dụng Persistent DAO để batch processing chạy tốt hơn.

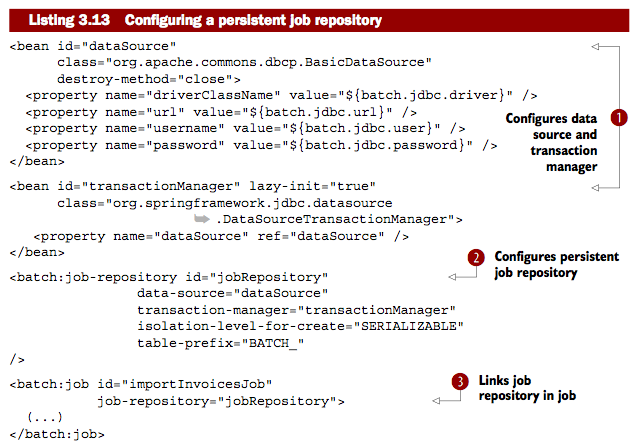
#### In-Memory with no persistent

* In-Memory sử dụng 2 class để configuration bao gồm:
  + MapJobRepositoryFactoryBean
  + ResourcelessTransactionManager



#### Persistent with metadata use JDBC

* Persistent sử dụng 2 class để configuration bao gồm:
  + JobRepositoryFactoryBean
  + DataSourceTransactionManager
* Những thuộc tính của JobRepository Persistent là:
  + data-source: dùng để truy xuất dữ liệu từ database, default:dataSource.
  + transaction-manager: dùng để xử lý các truy xuất dữ liệu của job repository, default:transactionManager.
  + isolation-level-for-create:
    - Tránh trường hợp 2 process cùng launch 1 job cùng 1 thời gian,tránh trường hợp 2 instance cùng tên,default:ISOLATION\_SERIALIZABLE.
    - Gồm 4 loại:
      * ISOLATION\_READ\_UNCOMMITTED
      * ISOLATION\_READ\_COMMITTED
      * ISOLATION\_REPEATABLE\_READ
      * ISOLATION\_SERIALIZABLE
    - REPEATABLE\_READ:cho phép 1 process launch 1 job và jobexecution trả success, nếu 2 process launch 1 job cùng 1 thời gian thì process thực thi trước sẽ có jobexecution trả về success,còn process thực thi sau sẽ có jobexecution trả về failure.
    - SERIALIZABLE: cho phép 2 process launch 1 job cùng 1 thời gian và cả 2 jobexecution của 2 process đều trả về success.
  + max-varchar-length: maximum length varchar colums in the database, default:2500.
  + table-prefix: xác định được tables job repository sử dụng trong database.
  + lob-handler: xử lý những cột trong database có chứa đối tượng dữ liệu lớn, thuộc tính này là tùy chọn, nó chỉ được sử dụng với Oracle hoặc khi Spring Batch không xác định được kiểu database .



1. JobLauncher

* JobLauncher sử dụng class configuration bao gồm:
  + SimpleJobLauncher
* Thuộc tính configuration của JobLauncher là:
  + jobRepository
  + taskExecutor
* TaskExecutor
  + TaskExecutor sử dụng class configuration bao gồm:
    - org.springframework.core.task.SimpleAsyncTaskExecutor.
    - thực thi các job bất đồng bộ(chạy các job cùng 1 lúc) để khi phát sinh ra lỗi thì nó có thể trả về ngay cho client.



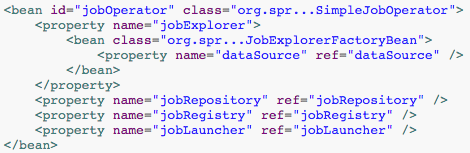
1. JobExplorer

* JobExplorer sử dụng class configuration bao gồm:
  + JobExplorerFactoryBean(Persistent)
  + MapRepository(In-Memory)
* Thuộc tính configuration của JobExplorer là:
  + data-source: dùng để truy xuất dữ liệu từ database, default:dataSource.

<beans:bean id="jobExplorer"  
class="org.springframework.batch.core.explore.support.JobExplorerFactoryBean" p:dataSource-ref="dataSource" />

### JobOperator

* JobOperator sử dụng class configuration bao gồm:
  + SimpleJobOperator
* Thuộc tính configuration của JobOperator là:
  + JobRepository
  + JobLauncher
  + JobExplorer
  + JobRegistry



### JobRegistry

* JobRegistry là quá trình ghi lại các instance của các job , sau đó JobOperator sẽ sử dụng những thông tin ghi lại đó để tìm ra job đã instance đó ⇒ đó gọi là track.
* JobRegistry sử dụng class configuration bao gồm:
  + MapJobRegistry

<bean id="jobRegistry" class="org.spr...MapJobRegistry" />

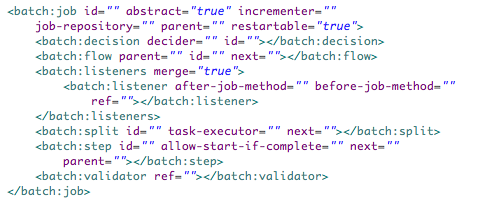
* JobRegistry gồm 2 cách configuration:
  + Bean Post Processor
  + Registry Lifecycle Compoment
* Bean Post Processor
  + Bean Post Processor sử dụng class configuration bao gồm:
    - org.springframework.batch.core.confirguration.support.MapRegistryBeanPostProcessor.

<bean id="jobRegistryBeanPostProcessor" class="org.spr...JobRegistryBeanPostProcessor">  
 <property name="jobRegistry" ref="jobRegistry"/>  
</bean>

* AutomaticJobRegistrar
  + AutomaticJobRegistrar sử dụng class configuration bao gồm:
    - ClassPathXmlApplicationContextFactory.
    - DefaultJobLoader

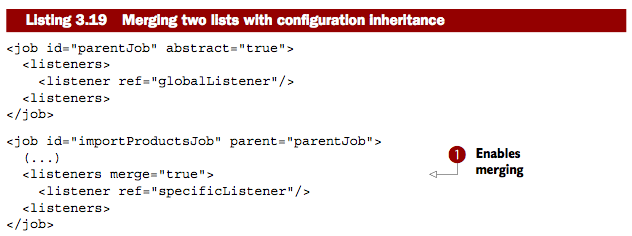


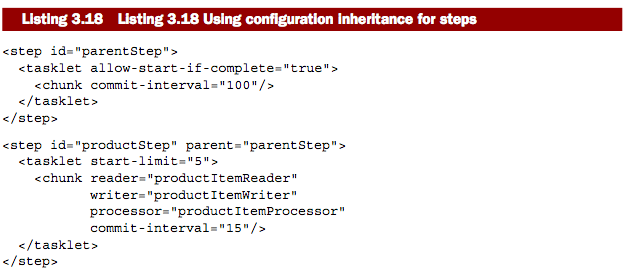
## Job



### Attributes

* id: định danh Job.
* restartable: cho phép Spring Batch có thể restart Job. df.
* abstract:configuration job là cha của các job khác nếu abstract=”true”, nó chứa những thuộc tính là:
  + listeners
  + validator

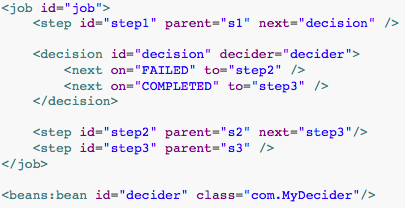


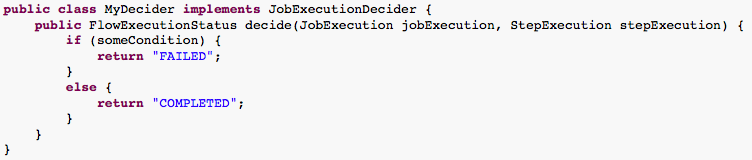
* parent: được định nghĩa như là lớp cha của Job.

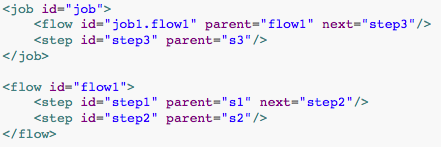
* Job Repository: định nghĩa job repository bean sử dụng cho jobRepository.

### Element

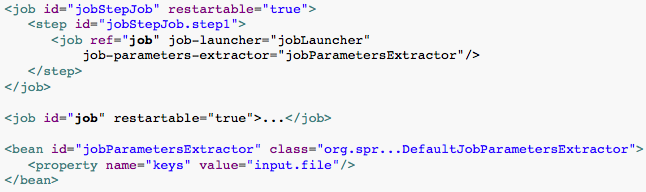
* decision: hỗ trợ quyết định thực hiện các step tiếp theo.



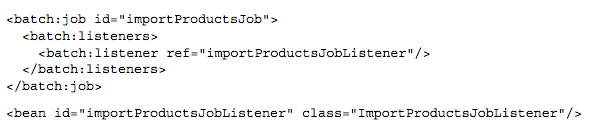


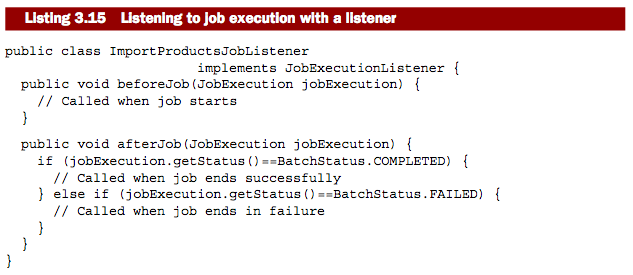
* flow :
  + Nó có thể tái sử dụng, cho nhiều job tham chiếu tới nó cùng 1 lúc để thực hiện những công việc khác nhau.
  + Có 2 cách thực hiện:
    - định nghĩa flow

* + - sử dụng flow như 1 jobStep

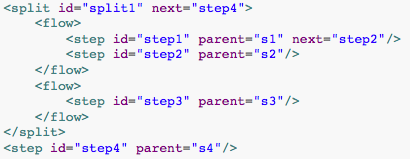


* listener : hỗ trợ trước và sau khi thực thi 1 job.



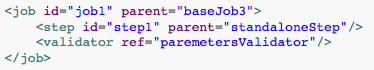


* spit(sub flow): thực hiện nhiều flow cùng 1 lúc.



* step: các bước thực hiện 1 Job

* validator: kiểm tra tất cả các thông số phải đúng với định dạng của Spring Batch.

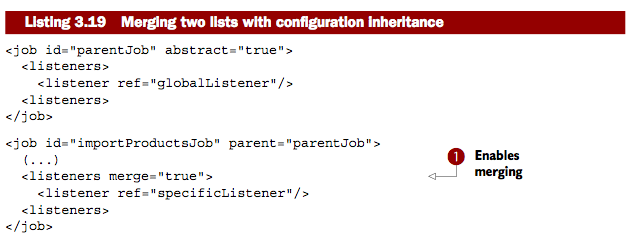


## Step

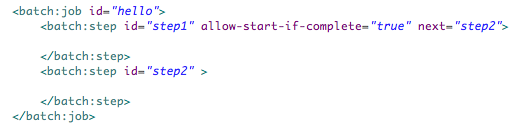
### Attributes



* next : xác định Step thực thi tiếp theo khi step này thực thi xong
* parent : step parent (step cha)
* abstract : nếu configuration abstract=”true” thì nó được xem như là Step parent.



* allow start if complete : nếu configuration allowStartIfComplete=”true” thì nếu Step đang thực thi complete thì mới thực thi next step, còn nếu fail thì step restart đến step thực thi bị failure.

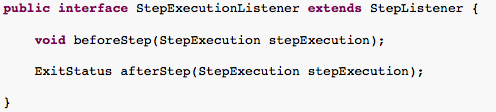


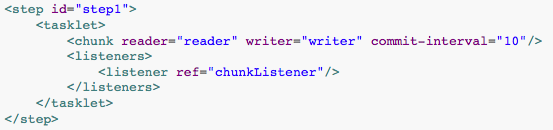
### Element

* tasklet : là quá trình lặp đi lặp lại việc xử lý dữ liệu trong 1 step.

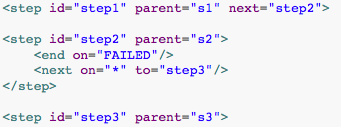


* listLiseners:
  + cho phép thông báo trước khi started step hoặc step thực thi xong dù có trả về status completed hoặc failure.
  + StepExecutionListener gồm 2 anotation:
    - @BeforeStep
    - @AfterStep





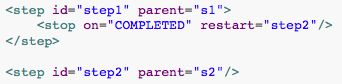
* end :
  + dùng để stop Job với BatchStatus=Completed.
  + không cho phép Job restart.
  + cho phép sử dụng thuộc tính “exit-code”.
  + nếu không có thuộc tính “exit-code” thì ExitStatus mặc định là “Completed” để trùng với BatchStatus.



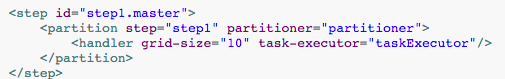
* fail :
  + dùng để stop Job với BatchStatus=Failed.
  + cho phép Job có thể restart.
  + cho phép sử dụng thuộc tính “exit-code” .
  + nếu không có thuộc tính “exit-code” thì ExitStatus mặc định là “Failed” để trùng với BatchStatus.



* stop:
  + dùng để stop Job với BatchStatus=STOPPED.
  + thoát khỏi step trước khi Job restart.
  + cần khai báo thuộc tính restart=”true” tại step cần thực thi lại sau khi Job restart.



* partion:
  + Spring Batch cung cấp scalling với partition
  + Ở level này, bạn có thể dùng multithreading and remoting.
  + Partition gồm 2 thành phần chính:
    - Data-partition: tạo ra các step execution.
    - Step Execution Handling:xử lý các execution khác nhau.Nó có thể là local(multithreading or not) và remote using Spring Inte-gration.

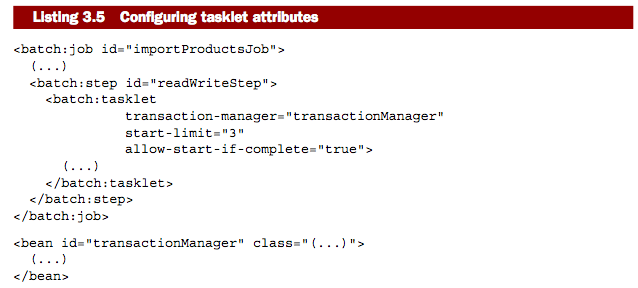




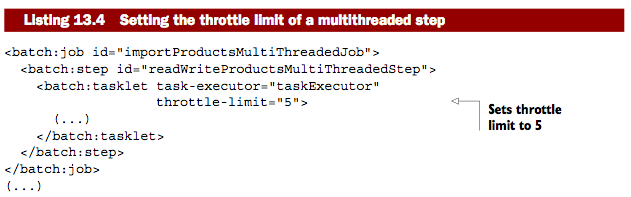
## Tasklet

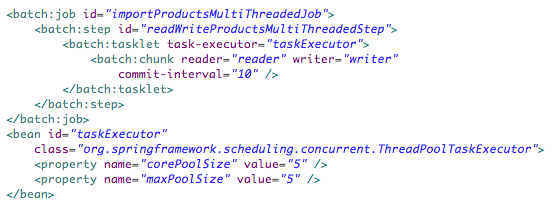
### Attributes

* ref: xác định class implements the Tasklet interface
* transaction-manager: Spring transaction manager sử dụng tasklet transactions. Default value:transactionManager.
* start-limit: thời gian Spring Batch có thể restart hoặc retry.
* allow-start-if-complete: cho phép Spring Batch có thể restart tasklet nếu retry nó sẽ trả về completed.



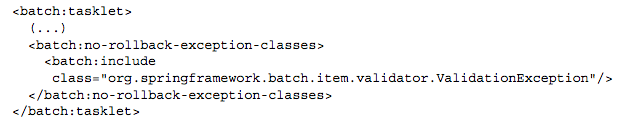
* throuttle-limit: hạn chế số tasklet khác nhau có thể thực thi cùng 1 thời gian.
* task-executor: các tasklet khác nhau thực thi cùng 1 thời gian.





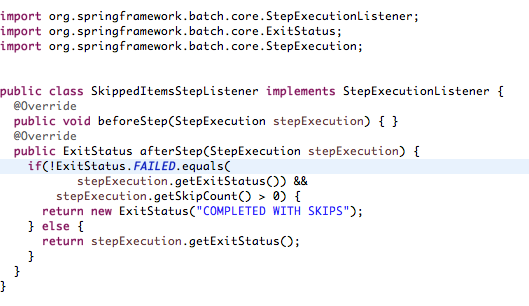
### Element

* no-rollback-exception-classes: tasklet sẽ không rollback khi xảy ra lỗi (classes exception).



* step execution listener:
  + Step execution listener là 1 trong nhiều listener.
  + Spring Batch gọi before and after execution của step execution listener.
  + Sau khi gọi lại after execution thì Step execution sẽ skip items (or not) và trả về exit status.
  + Tùy vào kết quả exit\_status trả về mà step execution listener sẽ quyết định thực thi step nào tiếp theo.
  + Step execution listener có thể đếm số lần skip trong tasklet.





## Chunk

### Attributes

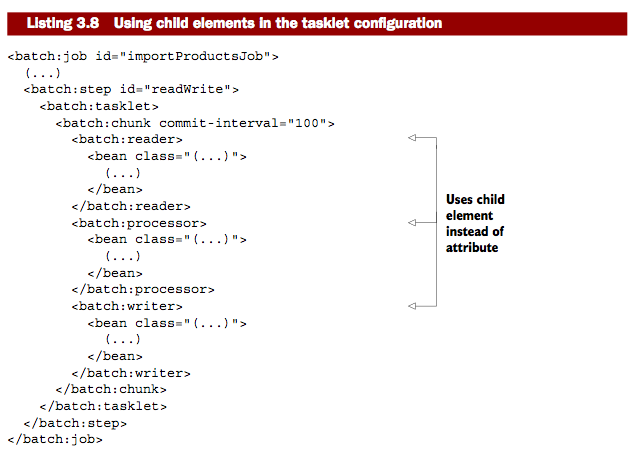
* reader: read data,,implement ItemReader interface.
* processor: process data, implement ItemProcessor interface.
* writer : write data, implement ItemWriter interface.
* commit-interval: số tập tin dữ liệu được read, processor,write trong 1 lần thực thi .



* skip-limit:maximum số lần skip trong chunk, nếu vượt quá số lần đó, step trả về fail.
* skip-policy: implement SkipPolicy interface.
* retry-limit:maximum số lần retries.
* cache-capacity: cache capacity of RetryPolicy.
* reader-transaction-quece: Khi reading items từ JMS quece, reading là transactional.
* processor-transacional: sử dụng processor bao gồm transaction processing.
* chunk-completion-policy:implement CompletionPolicy interface.

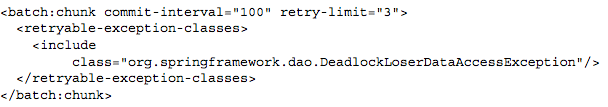


### Element

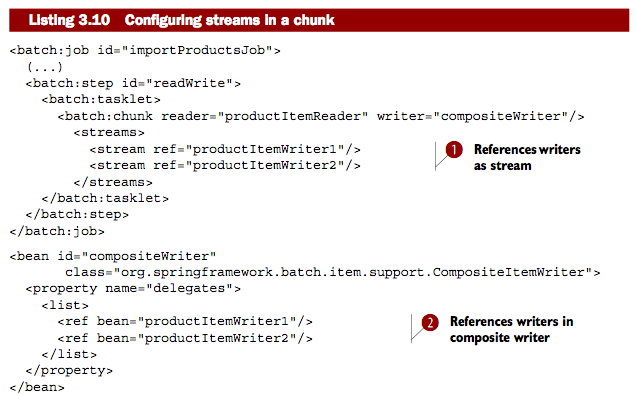


* chunk child elements
  + reader: tương ứng với thuộc tính reader.
  + processor: tương ứng với thuộc tính procesor.
  + writer:tương ứng với thuộc tính writer.
  + skip-policy:tương ứng với thuộc tính skip-policy.
  + retry-policy:tương ứng với thuộc tính retry-policy.
  + retry-listener
  + skippable-exception-classes: list exceptions triggering retries.

* + retryable-exception-classes:A list of exceptions triggering retries.



* + streams:
    - Tự động khai báo các thuộc tính reader,processor,writer.
    - implement ItemStream interface.
    - Giúp cho việc restart step thực thi nhanh hơn khi không phải return status execution.



# Readers, Writers, Processors

Spring Batch cung cấp 3 interfaces chính: ItemReader, ItemProcessor và ItemWriter, được dùng trong hầu hết các form đơn giản như khi đọc một lượng lớn dữ liệu, thực hiện các phép tính toán hay chuyển đổi…, và ghi ra kết quả.

## ItemReader

- Đọc dữ liệu từ nhiều kiểu dữ liệu khác nhau. VD:

· Flat File – FlatFileItemReader đọc các dòng dữ liệu từ file chứa các vùng dữ liệu có vị trí cố định hoặc được phân chia bởi các kí tự đặc biệt. (vd: dấu phẩy,…)

· XML – file xml mô tả dữ liệu.

· Database(mySql, SqlServer…)

- ItemReader interface:

|  |
| --- |
| **public interface** ItemReader<T> {  T read() **throws** Exception, UnexpectedInputException, ParseException;  } |

Hàm read() là quan trọng nhất, gọi hàm read() sẽ trả về một item hoặc null khi không có item. Một item có thể là: một line trong file, một row trong database hoặc một element trong file XML.

## ItemWriter

- ItemWriter interface:

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

Hàm write() xuất ra danh sách các item.

## ItemProcessor

- ItemProcessor interface:

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

- ItemProcessor nhận một đối tượng, chuyển hóa (transform) nó và trả về một đối tượng (object). Object trả về có thể cùng, hoặc không cùng kiểu.

VD: Một class kiểu Foo cần đổi sang class kiểu Bar trước khi được xuất ra.

|  |
| --- |
| **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  //Or do something else  **return new** Bar(foo);  }  }  **public class** BarWriter **implements** ItemWriter<Bar>{  **public void** write(List<? **extends** Bar> bars) **throws** Exception {  //write bars  }  } |

- Chú ý *FooProcessor* có thể tiêm(inject) vào trong một step

|  |
| --- |
| <job id="ioSampleJob">  <step name="step1">  <tasklet>  <chunk reader="fooReader" processor="fooProcessor" writer="barWriter"  commit-interval="2"/>  </tasklet>  </step>  </job> |

### Chaining ItemProcessors

- Một kỹ thuật để liên kết (xích) nhiều ItemProcessors lại với nhau.

|  |
| --- |
| **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  //Or do something else  **return new** Bar(foo);  }  }  **public class** BarProcessor **implements** ItemProcessor<Bar, FooBar>{  **public** FooBar process(Bar bar) **throws** Exception {  **return new** Foobar(bar);  }  }  **public class** FoobarWriter **implements** ItemWriter<Foobar>{  **public void** write(List<? **extends** FooBar> items) **throws** Exception {  //write items  }  } |

- FooProcessor và BarProcessor có thể được ‘liên kết’ (xích) lại với nhau-bằng *CompositeItemProcessor*- để trả về kết quả Foobar.

|  |
| --- |
| CompositeItemProcessor<Foo,Foobar> compositeProcessor =  new CompositeItemProcessor<Foo,Foobar>();  List itemProcessors = new ArrayList();  itemProcessors.add(new FooTransformer());  itemProcessors.add(new BarTransformer());  compositeProcessor.setDelegates(itemProcessors); |

- CompositeItemProcessor có thể được config trong Step.

|  |
| --- |
| <job id="ioSampleJob">  <step name="step1">  <tasklet>  <chunk reader="fooReader" processor="fooProcessor" writer="barWriter"  commit-interval="2"/>  </tasklet>  </step>  </job>  <bean id="compositeItemProcessor"  class="org.springframework.batch.item.support.CompositeItemProcessor">  <property name="delegates">  <list>  <bean class="..FooProcessor" />  <bean class="..BarProcessor" />  </list>  </property>  </bean> |

### Filtering Records

**-** Filtering Records: tạm dịch là các bản ghi được chọn lọc.

- Filtering(chọn lọc): là một hành động khác biệt với Skip. Skip chỉ ra rằng bản ghi (record) không hợp lệ, không thể được ghi ra. Còn Filtering chỉ ra record có thể hợp lệ, không nên được ghi ra hoặc được ghi ra dựa theo tiêu chí chọn lọc nào đó.

- VD: Một Job đọc một file chứa 3 kiểu records khác nhau: records “insert”, records “update”, records “delete”. Do hệ thống không hỗ trợ việc xóa records nên các records “delete” sẽ không được gửi đến *ItemWriter*. Nhưng các records “delete” này không thực sự là invalid, nên chúng ta muốn filter nó lại, không nên Skip. Và *ItemWriter* sẽ chỉ trả về các records “insert”, records “update”.

- Để filter một record, cách dễ nhất là return “null” trong *ItemProcessror*.

### Fault Tolerance

**-** Khi một chunk được rollback, các item đã được lưu trữ trong cache trong suốt quá trình reading có thể được reprocessed. Nếu một Step được config là fault tolerance (thường dùng Skip hoặc Retry), thì các *ItemProcessor* đã sử dụng nên được implement theo 1 cách được gọi là idempotent - không có sự thay đổi nào của các item được input vào *ItemProcessror* và chỉ cập nhật the instance..

## ItemStream

- Là một interface cần thiết. Trong batch Job, Readers và Writers cần được mở, đóng, trang bị một cơ chế nào đó.

|  |
| --- |
| public interface ItemStream {  void open(ExecutionContext executionContext) throws ItemStreamException;  void update(ExecutionContext executionContext) throws ItemStreamException;  void close() throws ItemStreamException;  } |

- ExecutionContext: là client của ItemReader và implement từ ItemStream. Nên gọi hàm open() trước khi gọi bất kỳ hàm read() nào đọc dữ liệu từ nguồn như file,.... Gọi hàm close() để đóng các resource khi open(). Hàm update() sẽ được gọi trước khi committing, và chắc chắn rằng current state được tạo trong database trước khi commit.

## The Delegate Pattern and Registering with the Step

**-** *CompositeItemWriter* là một ví dụ về Delegate Pattern, rất phổ biến trong Spring Batch.

- Nếu các delegates có implement interface StepListener, và đang dùng để kết hợp với Spring Batch Core trong một Step trong Job, vậy thì chúng cần phải đăng ký bằng tay (registered manually) với Step.

- Một Reader, Writer hoặc Processor được nối trực tiếp (directly wired) vào trong Step sẽ được đăng ký một cách tự động nếu nó implement ItemStream hoặc StepListener. Bởi vì các delegate không được hiểu trong Step, nên chúng cần được tiêm vào Step như một listeners hoặc streams( hoặc cả hai).

|  |
| --- |
| <job id="ioSampleJob">  <step name="step1">  <tasklet>  <chunk reader="fooReader" processor="fooProcessor" writer="compositeItemWriter"  commit-interval="2">  <streams>  <stream ref="barWriter" />  </streams>  </chunk>  </tasklet>  </step>  </job>  <bean id="compositeItemWriter" class="...CustomCompositeItemWriter">  <property name="delegate" ref="barWriter" />  </bean>  <bean id="barWriter" class="...BarWriter" /> |

## Flat Files

Là một trong những dạng lưu trữ dữ liệu phổ biến. Có 2 loại:

* Delimited files: bên trong chứa những vùng(field) dữ liệu được phân biệt.(Thông qua dấu phẩy “,”, …)
* Fixed Length files: chứa những vùng dữ liệu có độ dài cố định.

### The FieldSet

**-** Là một abstract class của Spring Batch để có thể binding dữ liệu từ 1 file resource (VD: file input, database input….), hỗ trợ việc parsing flat file input. FieldSet rất giống với Jdbc ResultSet. FieldSet chỉ yêu cầu 1 argument: một mảng String.

|  |
| --- |
| String[] tokens = new String[]{"foo", "1", "true"};  FieldSet fs = new DefaultFieldSet(tokens);  String name = fs.readString(0);  int value = fs.readInt(1);  boolean booleanValue = fs.readBoolean(2); |

### FlatFileItemReader

**-** Hỗ trợ việc reading và parsing flat file. Kết hợp với 2 interface quan trọng: *Resource* và *LineMapper*.

|  |
| --- |
| VD: Resource resource = new FileSystemResource("resources/trades.csv"); |

|  |  |  |
| --- | --- | --- |
| **Property** | **Type** | **Description** |
| comments | String[] | Specifies line prefixes that indicate comment rows |
| encoding | String | Kiểu text encoding muốn dùng - mặc định là "ISO-8859-1" |
| lineMapper | LineMapper | Converts 1 String sang 1 Object-đại diện cho item. |
| linesToSkip | int | Số lượng dòng bỏ qua (ignore) ở đầu file (top of file) |
| recordSeparatorPolicy | RecordSeparatorPolicy | Dùng để xác định nơi mà dòng(line) end và làm nhiều thứ (VD: tiếp tục qua dòng nếu bên trong 1 quoted string. |
| resource | Resource | Resource from which to read. |
| skippedLinesCallback | LineCallbackHandler | Interface này sẽ truyền raw line content (nội dung gốc) của những dòng(line) bị bỏ qua(skipped) trong file.  Nếu *linesToSkip* set là 2 thì interface sẽ được gọi 2 lần(twice). |
| strict | boolean | Trong chế độ ‘strict’(strict mode), Reader sẽ ném ra 1 exception trên *ExecutionContext* nếu input resource không tồn tại. |

Table F.1. FlatFileItemReader Properties

### LineMapper

**-** Convert a String line into an Object.

|  |
| --- |
| public interface LineMapper<T> {  T mapLine(String line, int lineNumber) throws Exception;  } |

### LineTokenizer

**-** Giúp chuyển đổi một hàng(line) của input thành một hàng trong FieldSet. Rất cần thiết vì có nhiều dạng dữ liệu flat file cần được convert vào FieldSet.

|  |
| --- |
| public interface LineTokenizer {  FieldSet tokenize(String line);  } |

- LineTokenizer’s implementations:

* DelimitedLineTokenizer: được dùng cho những file có các vùng dữ liệu phân biệt (thường là bởi dấu ‘|’ hoặc ‘;’, dấu’,’ …)
* FixedLengthTokenizer: được dùng cho những file có các vùng dữ liệu có chiều rộng cố định.
* PatternMatchingCompositeLineTokenizer: chứa 1 list LineTokenizers sẽ được dùng để kiểm tra theo Pattern.

### FieldSetMapper

- Interface FieldSetMapper có 1 hàm mapFieldSet(), để lấy 1 đối tượng FieldSet và map nó đến 1 đối tượng(a custom DTO, domain object, an array…). FieldSetMapper thường kết hợp cùng LineTokenizer để chuyển đổi dữ liệu từ một resource thành một đối tượng có kiểu như mong muốn.

|  |
| --- |
| public interface FieldSetMapper<T> {  T mapFieldSet(FieldSet fieldSet);  } //The pattern được dùng tương tự như RowMapper của JdbcTemplate. |

### DefaultLineMapper

**-** Từ việc định nghĩa các interface cho việc reading flat files ở trên. Chúng ta thấy có 3 bước cơ bản cần làm:

1. Đọc 1 dòng từ file.
2. Truyền chuỗi String vào hàm LineTokenizer#tokenize(), để chuyển thành FieldSet.
3. Truyền FieldSet trên vào FieldSetMapper, trả kết quả về từ hàm ItemReader#read().

- Khi input của LineTokenizer match(khớp) với input của LineMapper(a line), và output của FieldSetMapper match với output của LineMapper thì một default implement - sử dụng cả LineTokenizer và FieldSetMapper - được cung cấp. Đó chính là DefaultLineMapper

|  |
| --- |
| public class DefaultLineMapper<T> implements LineMapper<T>, InitializingBean {  private LineTokenizer tokenizer;  private FieldSetMapper<T> fieldSetMapper;  public T mapLine(String line, int lineNumber) throws Exception {  return fieldSetMapper.mapFieldSet(tokenizer.tokenize(line));  }  public void setLineTokenizer(LineTokenizer tokenizer) {  this.tokenizer = tokenizer;  }  public void setFieldSetMapper(FieldSetMapper<T> fieldSetMapper) {  this.fieldSetMapper = fieldSetMapper;  }  } |

### Exception Handling in Flat Files

**-** Spring Batch cung cấp một số lớp xử lý exception sau:

* FlatFileParseException: được thrown ra bởi FlatFileItemReader nếu xảy ra lỗi khi đang cố gắng đọc một file.
* FlatFileFormatException: được thrown ra bởi LineTokenizer interface nếu xảy ra lỗi khi tokenzing.
* IncorrectTokenCountException: Khi số column có thể dùng để tạo FieldSet không match với số column đã tìm thấy trong khi đang tokenizing một dòng(line) thì FieldSet không được tạo, exception được thrown ra.
* IncorrectLineLengthException:

## FlatFileItemWriter

**-** Write dữ liệu ra từ flat file có những vấn đề tương tự như Read.

### LineAggregator

**-** Giống như LineTokenizer interface, writing sẽ có cách kết hợp(aggregate) nhiều vùng dữ liệu thành một String và ghi ra file.

|  |
| --- |
| public interface LineAggregator<T> {  public String aggregate(T item);  } |

- LineTokenizer: lấy 1 String, return 1 FieldSet.

- LineAggregator: lấy 1 item, return 1 String.

### PassThroughLineAggregator

|  |
| --- |
| public class PassThroughLineAggregator<T> implements LineAggregator<T> {  public String aggregate(T item) {  return item.toString();  }  } |

**-** PassThroughLineAggregator: giả định rằng object đã là dạng String.

### FieldExtractor

- Các bước khi writing file:

* Truyền item cần write đến Writer.
* Convert các vùng dữ liệu(fields) trên item vào 1 mảng(array).
* Tập hợp(aggregate) mảng kết quả vào 1 dòng(line).

- Vì Spring Batch framework không có cách nào để biết được field từ object cần được ghi ra nên phải dùng FieldExtractor interface để hoàn thành việc chuyển đổi item vào trong 1 mảng.

|  |
| --- |
| public interface FieldExtractor<T> {  Object[] extract(T item);  } |

### PassThroughFieldExtractor

- Có nhiều trường hợp khi một collection(như là array, Collection, FieldSet) cần được ghi(write) ra. Việc giải nén(extracting) 1 mảng từ một trong những kiểu collection rất đơn giản: convert collection thành 1 mảng. Vì vậy, chúng ta nên sử dụng PassThroughFieldExtractor trong trường hợp này.Chú ý rằng khi 1 object truyền vào không phải là 1 kiểu collection thì PassThroughFieldExtractor sẽ trả về 1 mảng chỉ chứa các item đã được giải nén(extracted).

### BeanWrapperFieldExtractor

- Thường dùng để config cách convert 1 domain object sang 1 object array.

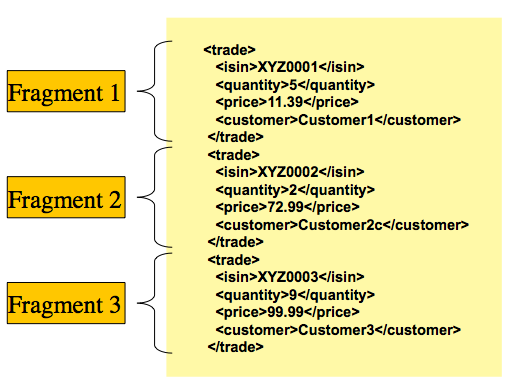
### Handling File Creation

- Reading file rất đơn giản: khởi tạo FlatFileItemReader, mở file muốn read, ném ra lỗi nếu file không tồn tại.

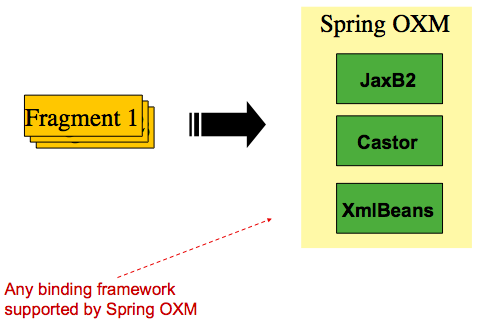
- Writing file thì không như thế:khởi tạo FlatFileItemWrite:

* TH1: nếu file đã tồn tại thì ném ra lỗi, nếu file không có thì tạo mới và bắt đầu ghi.
* TH2: Khi restart 1 Job, nếu file đã tồn tại thì bắt đầu ghi từ vị trí đã ghi gần đây nhất, nếu không thì ném ra lỗi.
* TH3: Khi file name trong 1 Job luôn luôn cùng 1 tên. Chúng ta sẽ muốn xóa file nếu file đã tồn tại. Dùng property shouldDeleteIfExists(true) của FlatFileItemWriter sẽ xóa file cùng tên khi mở writer.

## XML Item Readers and Writers



Hình G.1 - XML Input



Hình G.2 - XML và Spring OXM

### StaxEventItemReader

- Cung cấp cách cài đặt processing của các record từ XML input.

- Process các XML records cần:

* Root Element Name - Tên root element của fragment. VD: theo hình trên name = trade.
* Resource - đường dẫn đến file cần đọc.
* Unmarshaller - do Spring OXM hỗ trợ, mapping giữa XML fragment đến 1 object.

|  |
| --- |
| <bean id="itemReader" class="org.springframework.batch.item.xml.StaxEventItemReader">  <property name="fragmentRootElementName" value="trade" />  <property name="resource" value="data/iosample/input/input.xml" />  <property name="unmarshaller" ref="tradeMarshaller" />  </bean> |

### StaxEventItemWriter

- StaxEventItemWriter: cần có 1 Resource, 1 mashaller và 1 rootTagName. 1 Java object được truyền vào 1 mashaller (Spring OXM Marshaller)-ghi vào Resource bằng việc dùng custom event writer(event *StartDocument* và *EndDocument*) cho mỗi một fragment bằng OXM tools.

## Multi-File Input

- Giả sử có nhiều file có cùng formatting(txt, docx,...) trong 1 single Step. Class *MultiResourceItemReader*- có hỗ trợ các input dạng XML và flat file- có thể dùng để đọc các file.

|  |
| --- |
| <bean id="multiResourceReader" class="org.spr...MultiResourceItemReader">  // đọc tất cả các file có dạng file-\*.txt. VD: file-1.txt, file-2.txt...  <property name="resources" value="classpath:data/input/file-\*.txt" />  <property name="delegate" ref="flatFileItemReader" />  </bean> |

-

## Database

- Giả sử 1 câu lệnh SQL trả về 1 triệu rows, thì result set sẽ lưu toàn bộ kết quả trả về trong memory cho đến khi tất cả các rows được đọc. Spring Batch cung cấp 2 cách giải quyết vấn đề này: *Cursor* và *Paging Database ItemReaders*.

### Cursor Based ItemReaders

**-** Cursor(con trỏ) được mở khi khởi tạo cursor dựa trên ItemReaders. Cursor sẽ chuyển về phía trước 1 dòng(row) mỗi khi gọi read(), trả về 1 object dã được mapped - có thể được dùng cho processing. Sau đó close() sẽ được gọi để chắc rằng tất cả resources được giải phóng.

#### JdbcCursorItemReader

- Database:

|  |
| --- |
| CREATE TABLE CUSTOMER (  ID BIGINT IDENTITY PRIMARY KEY,  NAME VARCHAR(45),  CREDIT FLOAT  ); |

- Map 1 CustomerCredit object với RowMapper interface:

|  |
| --- |
| public class CustomerCreditRowMapper implements RowMapper {  public static final String ID\_COLUMN = "id";  public static final String NAME\_COLUMN = "name";  public static final String CREDIT\_COLUMN = "credit";  public Object mapRow(ResultSet rs, int rowNum) throws SQLException {  CustomerCredit customerCredit = new CustomerCredit();  customerCredit.setId(rs.getInt(ID\_COLUMN));  customerCredit.setName(rs.getString(NAME\_COLUMN));  customerCredit.setCredit(rs.getBigDecimal(CREDIT\_COLUMN));  return customerCredit;  }  } |

- Giả sử có 1000 rows trong CUSTOMER database. Khi ta dùng JdbcTemplate:

|  |
| --- |
| //For simplicity sake, assume a dataSource has already been obtained  JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  List customerCredits = jdbcTemplate.query("SELECT ID, NAME, CREDIT from CUSTOMER", new CustomerCreditRowMapper()); |

- Khi run code trên thì customerCredits list sẽ chứa 1000 CustomerCredit objects. Còn khi sử dụng JdbcCursorItemReader:

|  |
| --- |
| JdbcCursorItemReader itemReader = new JdbcCursorItemReader();  itemReader.setDataSource(dataSource);  itemReader.setSql("SELECT ID, NAME, CREDIT from CUSTOMER");  itemReader.setRowMapper(new CustomerCreditRowMapper());  int counter = 0;  ExecutionContext executionContext = new ExecutionContext();  itemReader.open(executionContext);  Object customerCredit = new Object();  while(customerCredit != null){  customerCredit = itemReader.read();  counter++;  }  itemReader.close(executionContext); |

- Khi run code trên, nếu trả về customerCredit list thì kết quả tương tự như khi dùng JdbcTemplate. Tuy nhiên, ItemReader cho phép items được “streamed”-(tạm dịch: xem trực tiếp). Hàm read() có thể được gọi 1 lần, item được ghi ra bởi ItemWriter và nhận item tiếp theo thu được qua read(). Việc reading và writing item được làm xong trong ‘chunks’ và được commited định kỳ. Đây chính là hiệu suất cao (high performance) của batch processing. Rất dễ tiêm(injection) vào Spring Batch Step:

|  |
| --- |
| <bean id="itemReader" class="org.spr...JdbcCursorItemReader">  <property name="dataSource" ref="dataSource"/>  <property name="sql" value="select ID, NAME, CREDIT from CUSTOMER"/>  <property name="rowMapper">  <bean class="org.springframework.batch.sample.domain.CustomerCreditRowMapper"/>  </property>  </bean> |

#### Additional Properties

|  |  |
| --- | --- |
| **Property** | **Description** |
| ignoreWarnings | Xác định có hay không SQLWarnings được logged hoặc gây ra 1 exception - default là true. |
| fetchSize | cho Jdbc driver gợi ý về số lượng rows nên được lấy(fetched) từ database khi mà có nhiều rows cần lấy bởi ResultSet object dùng ItemReader. |
| maxRows | giới hạn số lượng rows ResultSet có thể giữ tại cùng 1 thời điểm. |
| queryTimeout | Cài thời gian cho driver đợi 1 Statement object thực thi (excute). Nếu hết giờ, *DataAccessException* được ném ra. |
| verifyCursorPosition | Bởi vì các *ResultSet* giống nhau giữ bởi *ItemReader* được truyền vào *RowMapper*. Cho nên users có khả năng gọi hàm ResultSet.next(), có thể gây ra vấn đề với reader’s internal count. Cho value = true sẽ gây ra 1 exception nếu vị trí cursor không giống với vị trí sau khi *RowMapper* gọi cursor đó lúc trước. |
| saveState | Lưu tình trạng của reader (reader’s state) trong *ExecutionContext* cung cấp bởi *ItemStream#update(ExecutionContext)*.  Mặc định = true. |
| driverSupportsAbsolute | Mặc định = false.  Jdbc driver hỗ trợ các thiết lập dòng(row) tuyệt đối trên ResultSet.  Nếu value = true. Jdbc driver hỗ trợ: *ResultSet.absolute()*có thể cải thiện performance, ví dụ khi 1 Step fails trong khi đang làm việc với 1 lượng lớn dữ liệu. |
| setUseSharedExtendedConnection | Mặc định = false.  Cho biết connection dùng cho cursor nên được dùng bởi tất cả các processing khác thông qua việc sharing cùng transaction.  Nếu value = false, cursor sẽ được mở, dùng trong connection của riêng nó, và không share với bất kỳ transaction nào bắt đầu từ Step processing còn lại.  Nếu value = true, chúng ta phải bọc(wrap) *DataSource* trong *ExtendedConnectionDataSourceProxy* để ngăn connection bị đóng và giải phóng(released) sau mỗi commit. Câu lệnh (statement) đã dùng để mở cursor sẽ được tạo với cả 2 options ‘READ\_ONLY’ và ‘HOLD\_CURSORS\_OVER\_COMMIT’- cho phép giữ cursor mở khi transaction start và commit trong Step processing. (cần có database hỗ trợ Jdbc driver 3.0 hoặc hơn) |

#### HibernateCursorItemReader

- *HibernateCursorItemReader* cho phép chúng ta khai báo 1 câu lệnh HQL và truyền vào *SessionFactory* - sẽ truyền lại 1 item cho 1 lần gọi hàm read(), giống như JdbcCursorItemReader.

- VD:

|  |
| --- |
| HibernateCursorItemReader itemReader = new HibernateCursorItemReader();  itemReader.setQueryString("from CustomerCredit");  //For simplicity sake, assume sessionFactory already obtained.  itemReader.setSessionFactory(sessionFactory);  itemReader.setUseStatelessSession(true);  int counter = 0;  ExecutionContext executionContext = new ExecutionContext();  itemReader.open(executionContext);  Object customerCredit = new Object();  while(customerCredit != null){  customerCredit = itemReader.read();  counter++;  }  itemReader.close(executionContext); |

|  |
| --- |
| <bean id="itemReader"  class="org.springframework.batch.item.database.HibernateCursorItemReader">  <property name="sessionFactory" ref="sessionFactory" />  <property name="queryString" value="from CustomerCredit" />  </bean> |

#### StoredProcedureItemReader

- Chúng ta thực thi(execute) 1 stored procedure để trả về cursor. Có 3 cách(options):

1. như 1 *ResultSet* được trả về(dùng SQL Server, Sybase, DB2, Derby và MySQL)
2. như 1 ref-cursor trả về như 1 biến ngoài(out parameter). (dùng Oracle và PostgreSQL)
3. như 1 giá trị trả về của 1 stored function.

- VD: Options 1 : 1 *ResultSet*

|  |
| --- |
| <bean id="reader" class="o.s.batch.item.database.StoredProcedureItemReader">  <property name="dataSource" ref="dataSource"/>  <property name="procedureName" value="sp\_customer\_credit"/>  <property name="rowMapper">  <bean class="org.springframework.batch.sample.domain.CustomerCreditRowMapper"/>  </property>  </bean> |

- Option 2: return ref-cursor

|  |
| --- |
| <bean id="reader" class="o.s.batch.item.database.StoredProcedureItemReader">  <property name="dataSource" ref="dataSource"/>  <property name="procedureName" value="sp\_customer\_credit"/>  <property name="refCursorPosition" value="1"/>  <property name="rowMapper">  <bean class="org.springframework.batch.sample.domain.CustomerCreditRowMapper"/>  </property>  </bean> |

- Option 3: cursor được trả về từ 1 stored function

|  |
| --- |
| <bean id="reader" class="o.s.batch.item.database.StoredProcedureItemReader">  <property name="dataSource" ref="dataSource"/>  <property name="procedureName" value="sp\_customer\_credit"/>  <property name="function" value="true"/>  <property name="rowMapper">  <bean class="org.springframework.batch.sample.domain.CustomerCreditRowMapper"/>  </property>  </bean> |

- VD: Nếu stored procedure hoặc function lấy in-parameter(biến trong) thì chúng phải được khai báo và set(cài) thông qua property của parameter. VD cho Oracle dùng 3 parameter.

Parameter đầu tiên là out-parameter(biến ngoài) sẽ trả về ref-cursor, 2 parameter còn lại là in-parameter lấy giá trị có kiểu INTEGER.

|  |
| --- |
| <bean id="reader" class="o.s.batch.item.database.StoredProcedureItemReader">  <property name="dataSource" ref="dataSource"/>  <property name="procedureName" value="spring.cursor\_func"/>  <property name="parameters">  <list>  <bean class="org.springframework.jdbc.core.SqlOutParameter">  <constructor-arg index="0" value="newid"/>  <constructor-arg index="1">  <util:constant static-field="oracle.jdbc.OracleTypes.CURSOR"/>  </constructor-arg>  </bean>  <bean class="org.springframework.jdbc.core.SqlParameter">  <constructor-arg index="0" value="amount"/>  <constructor-arg index="1">  <util:constant static-field="java.sql.Types.INTEGER"/>  </constructor-arg>  </bean>  <bean class="org.springframework.jdbc.core.SqlParameter">  <constructor-arg index="0" value="custid"/>  <constructor-arg index="1">  <util:constant static-field="java.sql.Types.INTEGER"/>  </constructor-arg>  </bean>  </list>  </property>  <property name="refCursorPosition" value="1"/>  <property name="rowMapper" ref="rowMapper"/>  <property name="preparedStatementSetter" ref="parameterSetter"/>  </bean> |

### Paging ItemReaders

- Phân trang với *ItemReaders*: mỗi câu query đã được execute phải có row bắt đầu và số lượng row muốn trả về cho trang(page).

#### JdbcPagingItemReader

- Dùng *JdbcPagingItemReader* kết hợp với *SqlPagingQueryProviderFactoryBean*. VD:

|  |
| --- |
| <bean id="itemReader" class="org.spr...JdbcPagingItemReader">  <property name="dataSource" ref="dataSource"/>  <property name="queryProvider">  <bean class="org.spr...SqlPagingQueryProviderFactoryBean">  <property name="selectClause" value="select id, name, credit"/>  <property name="fromClause" value="from customer"/>  <property name="whereClause" value="where status=:status"/>  <property name="sortKey" value="id"/>  </bean>  </property>  <property name="parameterValues">  <map>  <entry key="status" value="NEW"/>  </map>  </property>  <property name="pageSize" value="1000"/>  <property name="rowMapper" ref="customerMapper"/>  </bean> |

#### JpaPagingItemReader

- *JpaPagingItemReader*: cho phép chúng ta khai báo 1 câu lệnh JPQL(Java Persistence Query Language) và truyền vào *EntityManagerFactory* - sẽ truyền lại 1 item cho 1 lần gọi hàm read(), giống như bất kỳ ItemReader khác.

|  |
| --- |
| <bean id="itemReader" class="org.spr...JpaPagingItemReader">  <property name="entityManagerFactory" ref="entityManagerFactory"/>  <property name="queryString" value="select c from CustomerCredit c"/>  <property name="pageSize" value="1000"/>  </bean> |

#### IbatisPagingItemReader (được yêu cầu-deprecated- dùng trong Spring Batch 3.0)

- Nếu chúng ta dùng IBATIS cho việc data access, ta có thể dùng IbatisPagingItemReader. IBATIS cung cấp một số standard variables giúp hỗ trợ phân trang ngay trong các câu query IBATIS.

|  |
| --- |
| <bean id="itemReader" class="org.spr...IbatisPagingItemReader">  <property name="sqlMapClient" ref="sqlMapClient"/>  <property name="queryId" value="getPagedCustomerCredits"/>  <property name="pageSize" value="1000"/>  </bean> |

- Câu query IBATIS ở trên đã gọi “getPagedCustomerCredits” có dạng vd sau(cho MySQL):

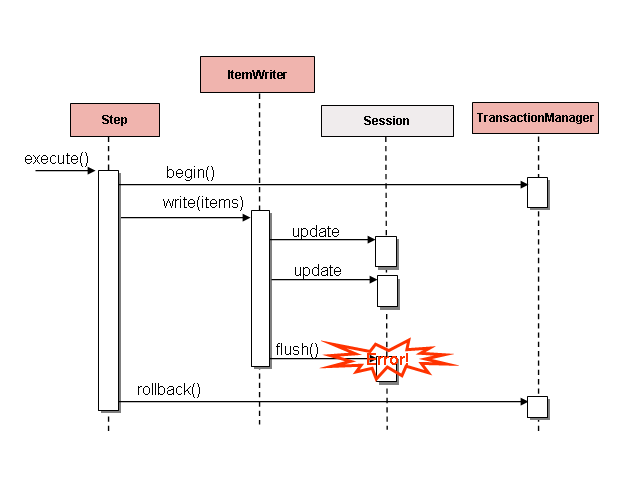
|  |
| --- |
| <select id="getPagedCustomerCredits" resultMap="customerCreditResult">  select id, name, credit from customer order by id asc LIMIT #\_skiprows#, #\_pagesize#  </select> |

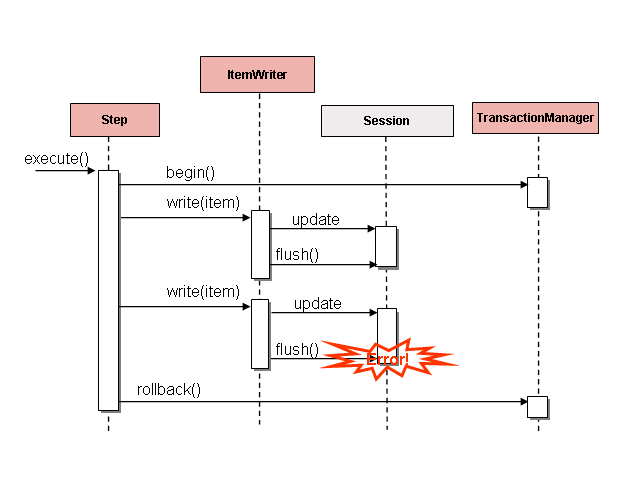
- *IbatisPagingItemReader*: cung cấp các biến: *\_skiprows*, *\_pagesize*, *\_page* sử dụng khi cần thiết. VD cho Oracle:

|  |
| --- |
| <select id="getPagedCustomerCredits" resultMap="customerCreditResult">  select \* from (  select \* from (  select t.id, t.name, t.credit, ROWNUM ROWNUM\_ from customer t order by id  )) where ROWNUM\_ <![CDATA[ > ]]> ( #\_page# \* #\_pagesize# )  ) where ROWNUM <![CDATA[ <= ]]> #\_pagesize#  </select> |

### Database ItemWriters

**-** Chỉ cần quan tâm đến hiệu quả (performance) và error handling bởi batching.



- VD như hình trên có 20 item, sau khi chúng ta update tất cả item không xảy ra lỗi, nhưng trong quá trình flush() thì lại xảy ra exception. Khi đó chúng ta sẽ khó xác định item nào gây ra exception. Cách xử lý duy trong trường hợp này là flush() từng item như hình dưới, khi đó chúng ta sẽ biết item nào gây ra exception.

## Reusing Existing Services

- Spring Batch cung cấp: *ItemReaderAdapter* và *ItemWriterAdapter* để có thể sử dụng lại các Services đã tồn tại.

- ItemReader:

|  |
| --- |
| <bean id="itemReader" class="org.springframework.batch.item.adapter.ItemReaderAdapter">  <property name="targetObject" ref="fooService" />  <property name="targetMethod" value="generateFoo" />  </bean>  <bean id="fooService" class="org.springframework.batch.item.sample.FooService" /> |

-ItemWriter tương ứng:

|  |
| --- |
| <bean id="itemWriter" class="org.springframework.batch.item.adapter.ItemWriterAdapter">  <property name="targetObject" ref="fooService" />  <property name="targetMethod" value="processFoo" />  </bean>  <bean id="fooService" class="org.springframework.batch.item.sample.FooService" /> |

## Validating Input

- Spring Batch cung cấp 1 interface đơn giản để validate: Validator interface

|  |
| --- |
| public interface Validator {  void validate(Object value) throws ValidationException;  } |

- VD box ItemProcessor:

|  |
| --- |
| <bean class="org.springframework.batch.item.validator.ValidatingItemProcessor">  <property name="validator" ref="validator" />  </bean>  <bean id="validator"  class="org.springframework.batch.item.validator.SpringValidator">  <property name="validator">  <bean id="orderValidator"  class="org.springmodules.validation.valang.ValangValidator">  <property name="valang">  <value>  <![CDATA[  { orderId : ? > 0 AND ? <= 9999999999 : 'Incorrect order ID' : 'error.order.id' }  { totalLines : ? = size(lineItems) : 'Bad count of order lines'  : 'error.order.lines.badcount'}  { customer.registered : customer.businessCustomer = FALSE OR ? = TRUE  : 'Business customer must be registered'  : 'error.customer.registration'}  { customer.companyName : customer.businessCustomer = FALSE OR ? HAS TEXT  : 'Company name for business customer is mandatory'  :'error.customer.companyname'}  ]]>  </value>  </property>  </bean>  </property>  </bean> |

## Preventing State Persistence

- Mặc định thì *ItemReader* và *ItemWriter* lưu trạng thái hiện tại (current state) vào *ExecutionContext* trước khi commited. Tuy nhiên, không phải lúc nào cũng như thế(có thể thay đổi tùy theo developers)

- Cấu hình : <bean>...**<property name=”saveState” value=”false”/>** …</bean>

## Creating Custom ItemReaders and ItemWriters

### Custom ItemReader

#### Example

|  |
| --- |
| public class CustomItemReader<T> implements ItemReader<T>{  List<T> items;  public CustomItemReader(List<T> items) {  this.items = items;  }  public T read() throws Exception, UnexpectedInputException,  NoWorkFoundException, ParseException {  if (!items.isEmpty()) {  return items.remove(0);  }  return null;  }  } |

|  |
| --- |
| List<String> items = new ArrayList<String>();  items.add("1");  items.add("2");  items.add("3");  ItemReader itemReader = new CustomItemReader<String>(items);  assertEquals("1", itemReader.read());  assertEquals("2", itemReader.read());  assertEquals("3", itemReader.read());  assertNull(itemReader.read()); |

#### Making the ItemReader Restartable

- Cấu hình CustomItemReader như sau:

|  |
| --- |
| public class CustomItemReader<T> implements ItemReader<T>, ItemStream {  List<T> items;  int currentIndex = 0;  private static final String CURRENT\_INDEX = "current.index";  public CustomItemReader(List<T> items) {  this.items = items;  }  public T read() throws Exception, UnexpectedInputException,  ParseException {  if (currentIndex < items.size()) {  return items.get(currentIndex++);  }  return null;  }  public void open(ExecutionContext executionContext) throws ItemStreamException {  if(executionContext.containsKey(CURRENT\_INDEX)){  currentIndex = new Long(executionContext.getLong(CURRENT\_INDEX)).intValue();  }  else{  currentIndex = 0;  }  }  public void update(ExecutionContext executionContext) throws ItemStreamException {  executionContext.putLong(CURRENT\_INDEX, new Long(currentIndex).longValue());  }  public void close() throws ItemStreamException {}  } |

- Mỗi khi ItemStream gọi update(), current index của ItemReader sẽ được lưu trong ExecutionContext với 1 key của ‘current.index’. Khi ItemStream gọi open(), Execution được check để xem có tồn tại 1 entry với key đó hay không, nếu có current index sẽ move đến vị trí đó.

|  |
| --- |
| ExecutionContext executionContext = new ExecutionContext();  ((ItemStream)itemReader).open(executionContext);  assertEquals("1", itemReader.read());  ((ItemStream)itemReader).update(executionContext);  List<String> items = new ArrayList<String>();  items.add("1");  items.add("2");  items.add("3");  itemReader = new CustomItemReader<String>(items);  ((ItemStream)itemReader).open(executionContext);  assertEquals("2", itemReader.read()); |

### Custom ItemWriter

#### Example

- VD:

|  |
| --- |
| public class CustomItemWriter<T> implements ItemWriter<T> {  List<T> output = TransactionAwareProxyFactory.createTransactionalList();  public void write(List<? extends T> items) throws Exception {  output.addAll(items);  }  public List<T> getOutput() {  return output;  }  } |

#### Making the ItemWriter Restartable

**-** Cùng 1 process như ItemReader.

# Parallelization configuration

## Multi-threaded:

* Một Step bình thường sẽ có định dạng như sau:

<batch:step id="processImages">

<batch:tasklet>

<batch:chunk reader="itemReader" processor="itemProcessor" writer="itemWriter" commit-interval="50"/>

</batch:tasklet>

</batch:step>

* Như đã đọc ở phần step configuration thì ở trên thì chắc không cần phải giải thích về phần ở trên nữa.
* Và nếu config như trên thì step trên sẽ chạy theo kiểu single thread.
* Nếu muốn chạy multi-threaded thì phải thêm “taskExecutor” vào trong config của Step. Dưới đây là một ví dụ thêm taskExecutor vào trong step:

<batch:step id="processImages">

<batch:tasklet task-executor="taskExecutor">

<batch:chunk reader="itemReader" processor="itemProcessor" writer="itemWriter" commit-interval="50"/>

</batch:tasklet>

</batch:step>

<bean id="taskExecutor" class="org.springframework.core.task.SimpleAsyncTaskExecutor">

<property name="concurrencyLimit" value="8"/>

</bean>

* Như bình thường thì dùng throttle-limit=”8” để limit số lượng thread, nhưng khi dùng taskExecutor với lớp SimpleAsyncTaskExecutor thì có thể dùng concurrencyLimit=”8” để limit số lượng thread có thể chạy đồng thời.
* Có thể xem thêm ở ví dụ ở đường link sau:

<https://numberformat.wordpress.com/2010/04/27/multi-threading-using-spring-batch/>

## Parallel step:

* Config parallel step rất đơn giản, chỉ thêm split và flow như sau



* Theo thứ tự chạy, thì các step setDataStep1, setDataStep2, setDataStep3 sẽ chạy parallel vs nhau trước, sau đó mới đến các step step1, step2, step3.

Screen Shot 2015-04-03 at 5.34.16 PM.png

* Bên cạnh đó còn phải config taskExecutor

## Remote chunking:

### Config master

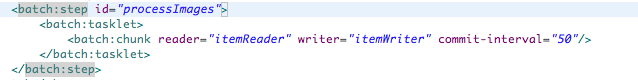
* Config máy chủ:



* bean=”connectionFactory” : define class ActiveMQConnectionFactory: define sử dụng máy chủ ActiveMQ
* bean=”messagingTemplate”: define class MesagingTemplate để gởi trả tin nhắn.
* Config step ở master:

Screen Shot 2015-04-02 at 4.33.35 PM.png

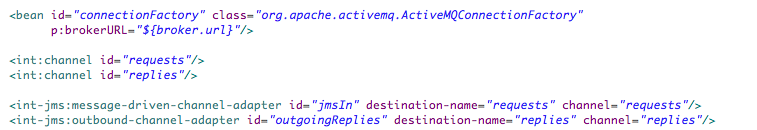
* Đây là config mới sẽ được release trong spring batch 3.1.0 khá là đơn giản: bao gồm các properties: itemWriter, mesageTemplate(lấy từ config mấy chủ), step ở slave.



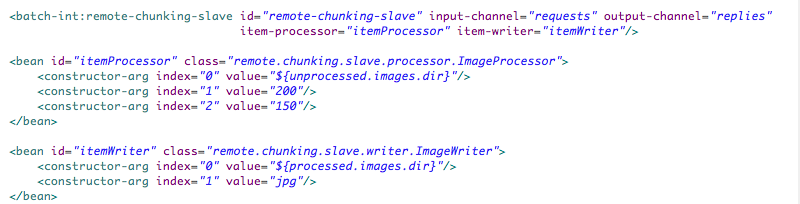
* Config step như bình thường, chỉ cần điền bean itemWriter cả batch-int:remote-chunking-master là được.

### Config slave:

* Config máy chủ: tương tự như ở master, chỉ cần config lăng nge server activeMQ là ok.



* Config step:



* Config step ở slave cũng rất đơn giản chỉ cần có batch-int:remote-chunking-slave để khai báo các input-channel dạng requests, output-channel là replies rồi điền các propertiese: itemProcessor và item-writer là được.

## Partitioning:

* Partitioning có thể chạy 2 loại là multi thread hoặc là remote node

### Partitioning multi thread:

* Ví dụ cấu hình partitioning chạy slave vs multi thread:

<job id="partitionJob" xmlns="http://www.springframework.org/schema/batch">

<step id="cleanup" next="masterStep">

<batch:tasklet ref="cleanupTasklet"/>

</step>

<!-- master step, 10 threads (grid-size) →

<step id="masterStep">

<partition step="slave" partitioner="rangePartitioner">

<handler grid-size="10" task-executor="taskExecutor" />

</partition>

</step>

</job>

<!-- each thread will run this job, with different stepExecutionContext values. -->

<step id="slave" xmlns="http://www.springframework.org/schema/batch">

<tasklet>

<chunk reader="pagingItemReader" writer="flatFileItemWriter"

processor="itemProcessor" commit-interval="1" />

</tasklet>

</step>

<bean id="rangePartitioner" class="com.test.spring.batch.partition.RangePartitioner" />

<bean id="taskExecutor" class="org.springframework.core.task.SimpleAsyncTaskExecutor" />

* Nhìn vào phần step=”masterStep” chúng ta chú ý vào thẻ “partition” nó sẽ gồm các attribute:
* Step=”slave”: sẽ gọi bean step có tên slave ở bên dưới, nhàm mục địch chia step này thành nhiều step nhỏ chạy cùng lúc với nhau.
* partitioner="rangePartitioner": gởi đến bean rangePartitioner ở bên dưới, bean rangePartitioner nhằm mục địch gọi lớp RangePartitioner(implements lớp Partitioner). Lớp RangePartitioner này sẽ nhận grip-size được quy định trong handler truyền vào hàm partition. Trong hàm partition chúng ta thực hiện truyền các data vào trong ExecutionContext sau đó return lại.
* grid-size="10": quy định số lượng thread có thể chạy.
* task-executor="taskExecutor": gọi bean taskExecutor
* Trong bean slave sẽ có tasklet và các attribute giống như một step bình thường như reader, writer, processor, commit-interval.

### Partitioning remote node:

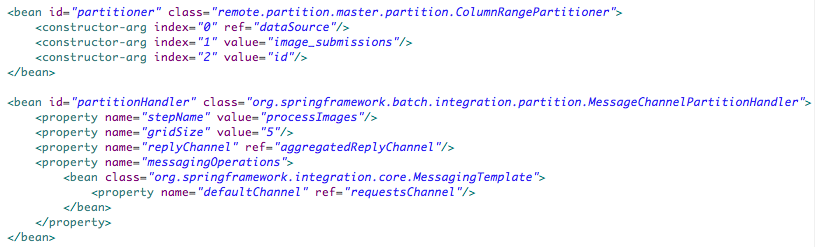
#### Config master

* Config máy chủ JMS:



* bean=”connectionFactory”: config using class ActiveMQConnectionFactory của mấy chủ ActiveMQ, p:brokerURL="${broker.url”: url of máy chủ ActiveMQ.
* int-jms:outbound-channel-adapter và int-jms:message-driven-channel-adapter không cần quan tâm tới, có thể đọc thêm ở spring integration.
* int:aggregator: có ref đến partitionHandler của job, sẽ được mô tả bên dưới.
* Config step có partition:

Screen Shot 2015-04-02 at 4.08.54 PM.png



* Trong step sẽ mô tả 2 bean là : partitioner và partitionHandler
* bean partitioner: dùng define classe ColumnRangPartitioner:đây là class implements Partitioner để custom dữ liệu truyền vào cho các slave vào các ExecutionContext.
* bean partitionHandler: define class MessageChannelPartitionHandler, dùng để giao tiếp vs slave. Trong đây sẽ define các property như
* stepName: đây là 1 bean define step ở slage
* gridsize: tối đã 5 partition or slave
* replyChannel: ref đến channel ở config máy chủ
* messagingOperations: ref đến channel ở config của máy chủ.

### Config slave

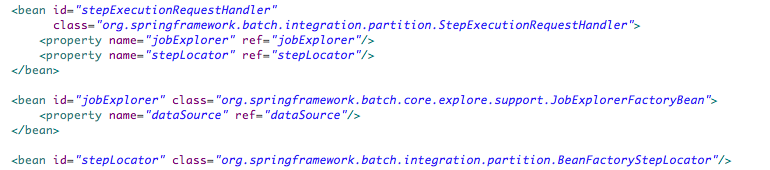
* Config máy chủ: tương tự như ở master như sau:



* Config slave:



* Ở trên bao gồm cá properties: itemReader, itemProcessor, itemWriter, coommit-interval: các properties này chắc hk cần phỏi nói thêm nữa, nên mình hk giải thích thêm, có thể đọc ở phần step configuration.



* bean stepExecutionRequestHandler dùng để ref trong config máy chủ, define class StepExecutionRequestHandler: dùng để handler StepExecutionRequest và trả về StepExecution.
* bean jobExplorer define class JobExplorerFactoryBean: dùng để check StepExecution.
* bean stepLocator: dùng để xác định vị trí của Step của mỗi request.

# Repeat and Retry

## Repeat:

### Repeat template:

* Spring batch cung cấp lớp RepeateOperations interface dùng để cung cấp số lượng repeate cho iterator framework. Đây là lớp chính dùng để truy cập batch operations(batch behavior, boundary condition, transaction).

|  |
| --- |
| public interface RepeatOperations {  RepeatStatus iterate(RepeatCallback callback) throws RepeatException;  } |

* RepeateCalback là interface cung cấp khả năng developer có thể thêm các business logic vào repeate.

|  |
| --- |
| public interface RepeatCallback {  RepeatStatus doInIteration(RepeatContext context) throws Exception;  } |

* Callback sẽ được chạy lặp lại nhiều lần cho đến khi việc triển khai quyết định vòng lặp nên dừng. Giá trị trả lại trong interface này là một enumeration có thể là RepeatStatus.CONTINUABLE hoặc RepeateStatus.FINISHED. RepeatStatus chuyển tải thông tin đến nới gọi thực hiện repeate operation về việc liệu có bất kì công việc nào cần thực hiện nữa hay hk. Đúng hơn là RepeateOperation nên kiểm trả xem RepeatStatus và dùng nó như là một phần của quyết định có nên tiếp tục or dừng vòng lặp. Bất kì Callback nào cũng mong muốn tín hiệu đến lớp gọi là không còn việc gì để làm (RepeateStatus.FINISHED)
* Sau đây là một ví dụ nhỏ về mục tiêu thực hiện của RepeateOperation là RepeateTemplate:

|  |
| --- |
| RepeatTemplate template = new RepeatTemplate();  template.setCompletionPolicy(new FixedChunkSizeCompletionPolicy(2));  template.iterate(new RepeatCallback() {  public ExitStatus doInIteration(RepeatContext context) {  // Do stuff in batch...  return ExitStatus.CONTINUABLE;  }  }); |

* Trong ví dụ, ta thấy nó return RepeateStatus.COTINUABLE để biểu thị là còn việc cần giải quyết. Callback có thể return ExitStatus.FINISHED nếu nó muốn tín hiệu trả về là không còn việc để làm.

#### RepeateContext:

* Method parameter của RepeatCallBack là RepeateContext.
* Rất nhiều CallBacks bỏ qua context, nhưng nếu cần nó có thể dùng attribute để lưu tạm thời data trong thời gian thực hiện iteration. Sau khi vòng lặp return, context sẽ được xoá.
* RepeateContext là parent context nếu có vòng lặp lồng nhau trong process. Parent context là đôi khi có lợi cho lưu trữ data mà data đó cần share giữ các iterate. Đây là trường hợp nếu bạn muốn đếm số lượng của xuất hiện của một sự kiển trong vòng lặp và nhớ nó thông qua lần gọi tiếp theo.

#### RepeataStatus:

* RepeateStatus là enumeration dùng trong Srping batch để xác định khi nào process hoàn thành. Có 2 status chính là
* Continuable: còn việc phải làm.
* Finished: không còn việc phải làm.

### Completion Policies:

* Trong RepeatContext, việc termination của loop trong iterate được xác định bởi CompletionPolicy nơi mà cũng là factory cho RepeateContext.
* RepeateTeamplate có trách nhiệm là dùng policy hiện tại để tạo RepeateContext và đưa nó vào RepeateCallback của mỗi nơi trong iterate.
* Sau khi Callback hoàn thành it trong doInIteration, RepeatTemplate phải tạo gọi đến CompletionPolicy để hỏi nó update state đó.
* Tiếp tục hỏi policy nếu iteration hoàn thành.
* Spring batch cung cấp một vài chức năng thực hiện trong CompletionPolicy. SimpleCompletionPolicy chỉ đồng ý thực hiện đến một con số nhất định của thời gian.
* Người dùng có thể cần triển khai thực hiện completion policies riêng của họ để thự hiện các quyết định phức tạp.

### Exception handling:

* Nếu như có exception thrown trong RepeateCallback, thì RepeateTemplate thông qua ExceptionHandler xem nó có quyết định có re-thrown exception hay không.

|  |
| --- |
| public interface ExceptionHandler {  void handleException(RepeatContext context, Throwable throwable)  throws RuntimeException;  } |

* Trường hợp phổ biến là đếm số lượng các exception của một loại nhất định và fail khi đạt đến giới hạn. Spring Batch cung cấp SimpleLimitExceptionHandler và RethrowOnThresholdExceptionHandler. SimpleLimitExceptionHandler có giới hạn các properrties và loại exception mà nên có để compared vs exception hiện tại. Exception của một loại nhất định được bỏ qua cho đến khi đạt đến giới hạn và được rethrown. Những loại khác luôn được rethrown.
* Một điều quan trọng trong optional property của SimpleLimitExceptionHandler là biến boolean flag useParent. Flase là giá trị mặc định của nó, vì vậy giới hạn chỉ tính trong RepeateContext hiện tại. Khi nó có giá trị true, giới hạn được giữ trong sibling contexts trong một vòng lặp lồng nhau.

### Listerners:

* Spring Batch cung cấp RepeatListener interface. RepeatTemplate cho phép user đăng ký RepeatListerner và nó cung cấp callback với RepeateContext và RepeateStatus nơi có hiện hữu iteration.
* RepeateListener có cấu trúc như sau:

|  |
| --- |
| public interface RepeatListener {  void before(RepeatContext context);  void after(RepeatContext context, RepeatStatus result);  void open(RepeatContext context);  void onError(RepeatContext context, Throwable e);  void close(RepeatContext context);  } |

* Open và close callback đến trước và sau khi iteration. Before, after, onError được áp dụng cho mỗi lần gọi RepeateCallback riêng lẻ.
* Chý ý là khi có nhiều hơn một listener, chúng trong 1 list, vì vậy nó là một thứ tự. Trong trường hợp này, open và before được gọi trong cùng một thứ tự trong khi after, onerror và close được gọi trong thứ tự đảo ngược.

### Parallel processing:

* RepeatOperation không giới hạn đến thực hiện callback sequentially. Nó khá quan trọng là trong vài trường hợp có thể được thực hiện callback trong parallel.
* Spring batch cung cấp lớp TaskExecutorRepeatemplate, nó sử dùng TaskExecutor để chạy RepeateCallback. Mặc định là SynchronousTaskExecutor, nó có hiệu quả trong thực hiện interation trong cùng một thread.

### Declarative iteration:

* Thỉnh thoảng có một vài business processing mà bạn muốn repeate mọi lần nó chạy. Ví dụ điều này là tối ưu hoá cho một message pipeline - nó rất có hiệu quả khi thực hiện một loạt các batch của mesage, nếu nó thực hiện thường xuyên hơn, và tài nguyên cho mỗi transaction thực hiện mỗi message cũng bị chiếm nhiều hơn. Spring batch cũng cấp AOP interceptor mà nó bao quanh method được gọi trong RepeatOperation. RepeatOperationInterceptor thực hiện intercepted method và repeat thông qua completionPolicy trong RepeatTeamplate.
* Sau dây là ví dụ về declaretive iteration sử dụng Spring AOP namespace để repeat service gọi đến method processMessage:

|  |
| --- |
| <aop:config>  <aop:pointcut id="transactional"  expression="execution(\* com..\*Service.processMessage(..))" />  <aop:advisor pointcut-ref="transactional"  advice-ref="retryAdvice" order="-1"/>  </aop:config>  <bean id="retryAdvice" class="org.spr...RepeatOperationsInterceptor"/> |

* Ví dụ bên trên sử dụng defaule RepeateTemplate trong interceptor. Để thay đổi policies, listener … bạn chỉ cần đưa vào một instance của RepeateTemplate trong interceptor.
* Nếu intercepted method trả về void thì interceptor luôn luôn return ExitStatus.CONTINUABLE (vì thế nên sễ rất nguy hiểm cho một vòng lặp vô hạng khi nếu CompletionPlicu không có điểm dừng hữu hạn. Nếu không nó trả về ExitStatus.CONTINUABLE cho đến khi trả về từ intercepted method là null, tại điểm mà nó trả về ExitStatus.FINISHED. Vì vậy business logic trong target method có thể đưa ra tín hiệu là không còn việc nào để thực hiện bằng việc return null hoặc thrown exception mà nó sẽ được re-thrown ExceptionHandler trong RepeatTeamplate.

## Retry:

### RetryTemplate

* Để tạo quá trình processing trở nên hoàn hảo và ít thất bại hơn, Spring batch cung cấp giải pháp Retry để tự động thử lại operation để nó có thể chạy thành công trong một nổ lực tiếp theo.
* Ví dụ như có remote call đến web service hoặc RMI service bị fail bởi vì do trục trặc network hoặc do DeadLoackLoserException trong database update có thể tự reslove sau một thời gian chờ.
* Để tự động Retry 1 operation, Spring Batch có RetryOperation. Nó có cấu trúc như sau:

|  |
| --- |
| public interface RetryOperations {    <T> T execute(RetryCallback<T> retryCallback) throws Exception;  <T> T execute(RetryCallback<T> retryCallback, RecoveryCallback<T> recoveryCallback) throws Exception;  <T> T execute(RetryCallback<T> retryCallback, RetryState retryState) throws Exception, ExhaustedRetryException;  <T> T execute(RetryCallback<T> retryCallback, RecoveryCallback<T> recoveryCallback, RetryState retryState) throws Exception;  } |

* Callback cănbanr là một interface cung cấp cho bạn khả năng thêm business logic để retry:

|  |
| --- |
| public interface RetryCallback<T> {    T doWithRetry(RetryContext context) throws Throwable;  } |

* Callback bị chạy và nếu nó fail, nó sẽ tự động try cho đến khi nó thành công hoặc khi quá limit retry thì việc triển khai bị huỷ bỏ. Có một số lượng overload chạy các method trong RetryOperations interface giao dịch vs các trường hợp khác nhau để recovery khi tất cả các retry đã đạt tới giới hạn số lượng và cũng với retry state, nó cũng cấp cho client và việc triển khai lưu trự thông tin giữa các lần gọi.
* Chức năng căn bản of RetryOperation là RetryTemplate. Nó có thể sử dụng như ví dụ sau:

|  |
| --- |
| RetryTemplate template = new RetryTemplate();  TimeoutRetryPolicy policy = new TimeoutRetryPolicy();  policy.setTimeout(30000L);  template.setRetryPolicy(policy);  Foo result = template.execute(new RetryCallback<Foo>() {  public Foo doWithRetry(RetryContext context) {  // Do stuff that might fail, e.g. webservice operation  return result;  }  }); |

* Trong ví dụ trên, nó gọi webservice và trả về result cho user, nếu như gọi fail, nó sẽ thử lại cho đến khi đạt đến timeout.

#### RetryContext

* Method parameter của RetryCallback là RetryContext. Nhiều callbacks sẽ bỏ quá context, nhưng nếu cần thiết nó có thể dùng attribute bag để lưu trữ data trong thời gian tồn tại của iteration.
* RetryContext sẽ có parent context nếu nó trong vòng retry lồng nhau (nested retry) trong quá trình trong cùng một thread. Trong một số trường hợp, Parent context có lợi trong lưu trữ data cần share giữa các lần gọi chạy.

#### RecoveryCallback

* Khi retry vượt quá limit thì RetryOperation có thể chuyển control đến callback khác, recoveryCallBack. Để dùng được chức năng này, client cần phải chuyển callback đồng thời trong cùng một method, ví dụ:

|  |
| --- |
| Foo foo = template.execute(new RetryCallback<Foo>() {  public Foo doWithRetry(RetryContext context) {  // business logic here  },  new RecoveryCallback<Foo>() {  Foo recover(RetryContext context) throws Exception {  // recover logic here  }  }); |

* Nếu business logic không thành công trước khi template quyết định huỷ bỏ, tiếp tục client được cho cơ hội để làm một số processing thay thế thông qua Recovery callback.

### Stateless Retry

* Trong trường hợp cụ thể, retry chỉ là trong vòng lặp: RetryTeamplate có thể chỉ giữ sự thử lại cho đến khi nó succeed hoặc fail. RetryContext chứa vài state để xác định có retry hay là huỷ bỏ quá trình, nhưng state này trên stack và nó không cần lưu ở globally, vì vậy nó được gọi là Stateless retry.
* Để phân biệt giữa stateless và stateful retry thì cần liên quan đến triển khai của RetryPolicy (RetryTemplate có thể handle cả 2).
* Trong stateless retry, callback luôn luôn được executed trong cùng thread trong thử lại giống khi nó fail.

### Stateful Retry

* Trong trường hợp sự thất bại gây ra gây ra transactional resource trở nên invalid, có một vài cân nhắc đặc biệt ở đây. Nó không được áp dụng cho remote call bởi vì nó không phải transactional resource, nhưng thỉnh thoảng áp dụng cho database update, đặc biêt khi sử dụng hibernate. Trong trường hợp này, nó chỉ có ý nghĩa với rethrow exception mà được gọi failure ngay lập tức vì vậy transaction có thể rollback và có thể start một new valid one.
* Trong nhiều trường hợp, stateless retry không đủ tốt bởi vì re-thrown và rollback cần thiết liên quan đến rời bỏ RetryOperations.execute methoad và có khả năng mất context có trong stack. Để tránh mất context, thì spring cung cấp khả năng lưu trữ bằng cách loại bỏ lưu trữ trong stack và lưu nó vào heap storage.
* Spring batch cung cấp RetryContextCache nó được thêm vào RetryTemplate. Mặc định triển khai của RetryContextCache là trong memory, sử dụng Map. Vấn đề nâng cao sử dụng với multi process ở môi trường clustered có thể cũng cân nhắc triển khai RetryContextCache với cluster cache của một số loại.
* Một phần trách nhiệm của RetryOperation là thừa nhận failed operation khi nó quay trở lại trong một execution mới (thường bao quan một transaction mới). Để tạo điều kiện thuận lợi, Spring batch cung cấp RetryState abstraction. Cái này kết hợp với execute method trong RetryOperations.
* Cái cách mà failed operation được thừa nhận là bởi xác định state băng qua multiple invocations của retry. Để xác định state, user có thể cung cấp RetryState object mà nó có trách nhiệm về sự trả về unique key xác định item. Đinh danh được sử dụng như một key trong RetryContextCache.
* Khi retry bị quá limit, nó cũng có option để handle item bị fail trong một cách khác, thay vì gọi RetryCalback. Giống như stateless case, option này được cung cấp bởi RecoveryCallback, nơi mà có thể cung cấp vượt qúa not trong việc execute mehod của retryOperation.
* Quyết định đến retry hoặc không thì thực tế uỷ quyền cho một RetryPolicy, vì vậy thường có mối quan tâm về limits và timeout có thể thêm vào đây.

### Retry Policies

* Bên trong RetryTeamplate, sự quyết định retry hoặc fail trong execute method bị xác định bởi RetryPolicy, nó cũng là factory của RetryContext.
* RetryTemplate có trách nhiệm tới việc dùng policy hiện tại để create RetryContext và pass cái đó vào bên trong RetryTemplate của mỗi attempt. Sau khi callback fail, RetryTemplate phải tạo lời gọi đến RetryPolicy để hỏi nó có update state đó (cái sẽ được lưu trong RetryContext), và tiếp tục hỏi policy nếu attempt khác có thể được tạo. Nêu attempt khác không thể được tạo (đạt đến limit hoặc timeout ) thì policy có trách nhiệm handle state này.
* Nếu bình thường vượt quá limit or timeout, nó sẽ throw ra RetryExhaustdException, kèm theo đó là cá transaction bị rolled back.
* Spring batch cung cấp một vài chức năng cơ bản cho việc triển khai stateless RetryPolicy, ví dụ như SimpleRetrypolicy và TimeoutRetryPolicy dùng trong ví dụ ở phải trên.
* SimpleRetryPolicy chỉ đồng ý cho retry một list loại các exception, và chạy trong một time nhất định. Nó cũng có list “fatal” exception, list fatal exception này không nên retry, và list này overrides retryable list, vì vậy nó có thể bị sử dụng để cho quản lý tốt hơn retry behavior. Ví dụ

|  |
| --- |
| SimpleRetryPolicy policy = new SimpleRetryPolicy();  // Set the max retry attempts  policy.setMaxAttempts(5);  // Retry on all exceptions (this is the default)  policy.setRetryableExceptions(new Class[] {Exception.class});  // ... but never retry IllegalStateException  policy.setFatalExceptions(new Class[] {IllegalStateException.class});  // Use the policy...  RetryTemplate template = new RetryTemplate();  template.setRetryPolicy(policy);  template.execute(new RetryCallback<Foo>() {  public Foo doWithRetry(RetryContext context) {  // business logic here  }  }); |

* Sẽ linh hoạt hơn nếu gọi ExceptiponClassifierRetryPolicy, nó cung cấp cho user khả năng config các retry behavior khác nhau cho các thiết lập tuỳ ý của các loại exception bên trong RetryPolicy. Ví dụ như: một loại exception có thể retry nhiều lần trước khi fail bằng mapping nó đến khác policy khác.
* Người dùng có thể cần implement retry policy của chính mình.

### Backoff policies

* Khi đang retry sau khi tạm thời quá trình bị fail, backoff policies giúp quá trình chờ một khoảng thời gian rồi mới retry trở lại. Bởi vì thông thường fail thường có lí bởi một vài vấn đề mà nó có thể dược giải quyết khi chờ một thời gian. Nếu RetryCallback fail, thì RetryTemplate có thể pause execution tuỳ theo BackoffPolicy.
* Backoffpolicy interface trông như sau:

|  |
| --- |
| public interface BackoffPolicy {  BackOffContext start(RetryContext context);  void backOff(BackOffContext backOffContext) throws BackOffInterruptedException;  } |

### Listeners

* Spring batch cùng cấp RetryListener interface.
* Trong RetryTemplate cho phép user thêm RetryListeners, và nó sẽ bị cho callBack với RetryContext và Throwable nới hiện hữu trong quá trình iteration.
* RetryListeners interface như sau:

|  |
| --- |
| public interface RetryListener {  void open(RetryContext context, RetryCallback<T> callback);  void onError(RetryContext context, RetryCallback<T> callback, Throwable e);  void close(RetryContext context, RetryCallback<T> callback, Throwable e);  } |

* Open và close method đến trước và sau retry trong một trường hợp đơn giản nhất và onerror được áp dùng để RetryCallback gọi. Close method có thể nhận Throwable, nêu nó có error nó sẽ là được thrown bởi RetryCallback.

### Declarative Retry

* Thỉnh thoảng, có một vài business processing mà người dùng biết là họ muốn nó chạy mọi lần trong retry. Ví dụ cơ bản về cái này là Remote service call.
* Spring batch cung cấp AOP interceptor bao quanh method được gọi trong RetryOperations cho mục đích này.
* RetryOperationInterceptor chạy intercepter method và retry khi fail thông qua RetryPolicy được cung cấp bởi RepeatTemplate.
* Sau đây là một example cho khai báo iteration sử dùng Spring AOP namespace để lăp lại service call đến method called remoteCall.

|  |
| --- |
| <aop:config>  <aop:pointcut id="transactional"  expression="execution(\* com..\*Service.remoteCall(..))" />  <aop:advisor pointcut-ref="transactional"  advice-ref="retryAdvice" order="-1"/>  </aop:config>  <bean id="retryAdvice"  class="org.springframework.batch.retry.interceptor.RetryOperationsInterceptor"/> |

* Ví dụ bên trên sử dụng RetryTemplate bên trong interceptor. Để cthay đổi policies hoặc listeners, bạn chỉ cần thêm các instance của RetryTemplate bên trong interceptor.

# Logging

**- Common Batch Patterns:** **dưới đây là một vài ví dụ về các pattern phổ biến trong custom business logic(cho phép sử dụng các listener interfaces).**

## Logging Item Processing and Failures

- Cần dùng trong việc xử lý lỗi trong 1 Step, từng item một, có lẽ nên ghi(logging) vào 1 kênh(channel) đặc biệt hoặc insert 1 record vào trong database. 1 chunk-oriented Step(tạo ra từ Step factory beans) cho phép users sử dụng *ItemReadListener-cho việc đọc lỗi* và *ItemWriteListener-cho việc ghi lỗi*.

|  |
| --- |
| public class ItemFailureLoggerListener extends ItemListenerSupport {  private static Log logger = LogFactory.getLog("item.error");  public void onReadError(Exception ex) {  logger.error("Encountered error on read", e);  }  public void onWriteError(Exception ex, Object item) {  logger.error("Encountered error on write", e);  }  } |

- Sau khi sử dụng implement trên thì chúng ta phải register listener vào Step

|  |
| --- |
| <step id="simpleStep">  ...  <listeners>  <listener>  <bean class="org.example...ItemFailureLoggerListener"/>  </listener>  </listeners>  </step> |

- Hãy nhớ rằng nếu listener làm bất kỳ điều gì trong 1 hàm onError(), nó sẽ ở bên trong 1 transaction-sẽ bị rollback lại. Nếu chúng ta cần dùng 1 transactional resource ví dụ như database bên trong hàm onError() thì chúng ta phải thêm vào 1 declarative transaction cho hàm đó và propagation attribute giá trị = REQUIRES\_NEW.

## Stopping Job Manually for Business Reasons

- Spring Batch cung cấp 1 hàm stop() thông qua *JobLauncher* interface, nhưng đôi khi stop 1 Job từ bên trong business logic tiện lợi và hợp lý hơn cả.

- Cách đơn giản nhất là ném ra RuntimeException.VD:

|  |
| --- |
| public class PoisonPillItemWriter implements ItemWriter<T> {  public void write(T item) throws Exception {  if (isPoisonPill(item)) {  throw new PoisonPillException("Posion pill detected: " + item);  }  }  } |

- Một cách đơn giản khác để stop 1 Step từ việc thực thi(executing) là return *null* từ *ItemReader*.

|  |
| --- |
| public class EarlyCompletionItemReader implements ItemReader<T> {  private ItemReader<T> delegate;  public void setDelegate(ItemReader<T> delegate) { ... }  public T read() throws Exception {  T item = delegate.read();  if (isEndItem(item)) {  return null; // end the step here  }  return item;  }  } |

- Ví dụ trên cho biết thật ra là có 1 default implement của *CompletionPolicy* - đánh dấu 1 complete batch khi item bị processed là *null*. 1 *ComplationPolicy* có thể được implemented và injected vào trong Step thông qua *SimpleStepFactoryBean*.

|  |
| --- |
| <step id="simpleStep">  <tasklet>  <chunk reader="reader" writer="writer" commit-interval="10"  chunk-completion-policy="completionPolicy"/>  </tasklet>  </step>  <bean id="completionPolicy" class="org.example...SpecialCompletionPolicy"/> |

- Một cách khác là đặt 1 flag trong *StepExecution* - cái mà được check bởi Step implements trong framework trong giữa item processing. Để dùng cách này, chúng ta cần phải access đến *StepExecution* hiện tại, bằng cách implement 1 *StepListener* và register nó với Step. VD:

|  |
| --- |
| public class CustomItemWriter extends ItemListenerSupport implements StepListener {  private StepExecution stepExecution;  public void beforeStep(StepExecution stepExecution) {  this.stepExecution = stepExecution;  }  public void afterRead(Object item) {  if (isPoisonPill(item)) {  stepExecution.setTerminateOnly(true);  }  }  } |

- Mặc định ở đây khi flag được set là cho Step ném ra *JobInterruptedException*. Có thể được điều khiển(controlled) thông qua *StepInterruptionPolicy*.

## Adding a Footer Record

**-** Thông thường khi writing 1 flat file, 1 “footer” record phải được thêm vào cuối file, sau khi tất cả processing đã được completed. Có thể dùng *FlatFileFooterCallback* interface của Spring Batch. *FlatFileFooterCallback ( và FlatFileHeaderCallback)* là properties tùy chọn của *FlatFileItemWriter*:

|  |
| --- |
| <bean id="itemWriter" class="org.spr...FlatFileItemWriter">  <property name="resource" ref="outputResource" />  <property name="lineAggregator" ref="lineAggregator"/>  <property name="headerCallback" ref="headerCallback" />  <property name="footerCallback" ref="footerCallback" />  </bean> |

- Footer callback interface rất đơn giản. Nó chỉ có 1 hàm được gọi khi footer được viết:

|  |
| --- |
| public interface FlatFileFooterCallback {  void writeFooter(Writer writer) throws IOException;  } |

### Writing a Summary Footer

- Một cách phổ biến khi dùng footer records là tổng hợp information trong suốt quá trình output process và sau đó thêm(append) information này vào cuối file.

- VD: Nếu batch job đang writing Trade records vào 1 flat file, thì có 1 biến ghi lại tổng cộng (total amount) tất cả Trades được write trong footer.

|  |
| --- |
| public class TradeItemWriter implements ItemWriter<Trade>,  FlatFileFooterCallback {  private ItemWriter<Trade> delegate;  private BigDecimal totalAmount = BigDecimal.ZERO;  public void write(List<? extends Trade> items) {  BigDecimal chunkTotal = BigDecimal.ZERO;  for (Trade trade : items) {  chunkTotal = chunkTotal.add(trade.getAmount());  }  delegate.write(items);  // After successfully writing all items  totalAmount = totalAmount.add(chunkTotal);  }  public void writeFooter(Writer writer) throws IOException {  writer.write("Total Amount Processed: " + totalAmount);  }  public void setDelegate(ItemWriter delegate) {...}  } |

- *TradeItemWriter* ở trên lưu 1 giá trị totalAmount sẽ tăng dần mỗi khi có 1 Trade item được ghi. Sau khi Trade cuối cùng được processed thì framework sẽ gọi writeFooter()-sẽ đặt(put) totalAmount vào trong file.

- Để writeFooter() được gọi, thì *TradeItemWriter* phải được wired vào trong *FlatFileItemWriter*:

|  |
| --- |
| <bean id="tradeItemWriter" class="..TradeItemWriter">  <property name="delegate" ref="flatFileItemWriter" />  </bean>  <bean id="flatFileItemWriter" class="org.spr...FlatFileItemWriter">  <property name="resource" ref="outputResource" />  <property name="lineAggregator" ref="lineAggregator"/>  **<property name="footerCallback" ref="tradeItemWriter" />**  </bean> |

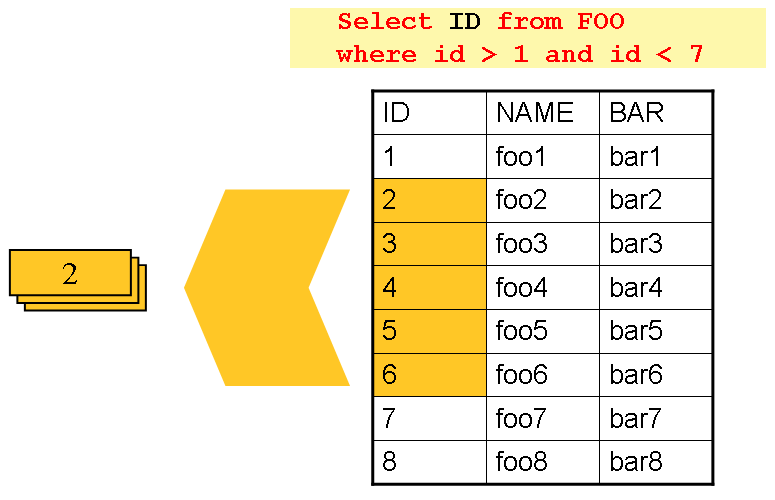
- Cách dùng class *TradeItemWriter* chỉ đúng khi mà Step không phải restartable. Bởi vì class là stateful(do nó lưu *totalAmount*), nhưng *totalAmount* không được tạo trong database nên không thể khôi phục khi xảy ra event restart. Để làm cho class này restartable thì *ItemStream* interface nên được implemented cùng với hàm open() và update():

|  |
| --- |
| public void open(ExecutionContext executionContext) {  if (executionContext.containsKey("total.amount") {  totalAmount = (BigDecimal) executionContext.get("total.amount");  }  }  public void update(ExecutionContext executionContext) {  executionContext.put("total.amount", totalAmount);  } |

- Hàm update() sẽ lưu version hiện tại mới nhất của *totalAmount* đến *ExecutionContext* trước khi object đó được tạo(persisted) trong database. Hàm open() sẽ khôi phục *totalAmount* đã tồn tại từ *ExecutionContext* và dùng nó tại điểm bắt đầu processing, cho phép *TradeItemWriter* chọn nó khi restart Step.

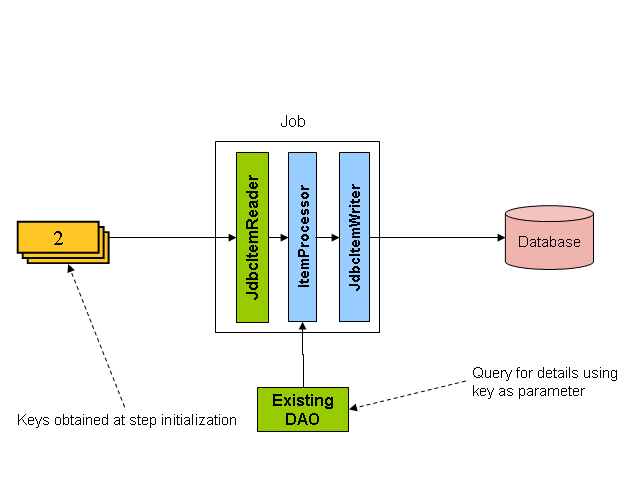
## Driving Query Based ItemReaders

- Có nhiều projects thích sử dụng ‘Driving Query’ cho việc reading in data. Thường làm việc với key(khóa chính,...) hơn là với cả 1 object-mà cần được trả về.



Hình VIII. - Driving Query Example

- Từ hình trên ta thấy chỉ có ID được select trong SQL statement, 1 Integer được trả về thay vì cả 1 FOO object. Số (Integer) này có thể được dùng cho câu query “detail”-sẽ trả về 1 FOO object.



Hình VIII. - Driving Query Job

- 1 *ItemProcessor* nên được dùng để chuyển đổi(transform) key thành 1 full ‘Foo’ object.

## Multi-Line Records

- Thông thường với flat file: mỗi một record được confined(tạm dịch: ghi) vào 1 single line. Nhưng có những dạng file phổ biến: với nhiều records có multi line với multi formats. VD:

|  |
| --- |
| HEA;0013100345;2007-02-15  NCU;Smith;Peter;;T;20014539;F  BAD;;Oak Street 31/A;;Small Town;00235;IL;US  FOT;2;2;267.34 |

- Mọi thứ ở giữa dòng(line) bắt đầu với “HEA” và dòng bắt đầu với”FOT” là 1 record.

* Thay vì đọc 1 record tại 1 thời điểm, *ItemReader* phải đọc từng line của multi-line record như là 1 group, để có thể truyền vào *ItemWriter*
* Từng loại line(line type) có thể cần được tokennized khác nhau.

- Bởi vì 1 single record chứa nhiều multiple lines, và chúng ta có thể không biết có bao nhiêu lines nên *ItemReader* phải luôn cẩn thận khi đọc cả 1 record. Để làm điều đó, 1 custom *ItemReader* nên được implemented như 1 wrapper cho *FlatFileItemReader*

|  |
| --- |
| <bean id="itemReader" class="org.spr...MultiLineTradeItemReader">  <property name="delegate">  <bean class="org.springframework.batch.item.file.FlatFileItemReader">  <property name="resource" value="data/iosample/input/multiLine.txt" />  <property name="lineMapper">  <bean class="org.spr...DefaultLineMapper">  <property name="lineTokenizer" ref="orderFileTokenizer"/>  <property name="fieldSetMapper">  <bean class="org.spr...PassThroughFieldSetMapper" />  </property>  </bean>  </property>  </bean>  </property>  </bean> |

- Để chắc chắn rằng mỗi line được tokenized thì có thể dùng *PatternMatchingCompositeLineTokenizer* trên *FlatFileItemReader*. Reader sẽ dùng *PassThroughFieldSetMapper* để giao(deliver) 1 *FieldSet* cho mỗi line quay lại the wrapping ItemReader.

|  |
| --- |
| <bean id="orderFileTokenizer" class="org.spr...PatternMatchingCompositeLineTokenizer">  <property name="tokenizers">  <map>  <entry key="HEA\*" value-ref="headerRecordTokenizer" />  <entry key="FOT\*" value-ref="footerRecordTokenizer" />  <entry key="NCU\*" value-ref="customerLineTokenizer" />  <entry key="BAD\*" value-ref="billingAddressLineTokenizer" />  </map>  </property>  </bean> |

- Wrapper trên sẽ có thể nhận ra kết thúc(the end) của record, nên nó có thể tiếp tục gọi hàm read() cho đến khi đến the end. Cho mỗi line được đọc, wrapper nên xây dựng trên(build up) item được trả về. Một khi đạt tới footer thì item có thể được return cho *ItemProcessor* và *ItemWriter*.

|  |
| --- |
| private FlatFileItemReader<FieldSet> delegate;  public Trade read() throws Exception {  Trade t = null;  for (FieldSet line = null; (line = this.delegate.read()) != null;) {  String prefix = line.readString(0);  if (prefix.equals("HEA")) {  t = new Trade(); // Record must start with header  }  else if (prefix.equals("NCU")) {  Assert.notNull(t, "No header was found.");  t.setLast(line.readString(1));  t.setFirst(line.readString(2));  ...  }  else if (prefix.equals("BAD")) {  Assert.notNull(t, "No header was found.");  t.setCity(line.readString(4));  t.setState(line.readString(6));  ...  }  else if (prefix.equals("FOT")) {  return t; // Record must end with footer  }  }  Assert.isNull(t, "No 'END' was found.");  return null;  } |

## Executing System Commands

- Nhiều batch job có thể yêu cầu 1 command bên ngoài được gọi từ bên trong batch job. VD: 1 multi-step job cần được chia ra nhiều multi job.

- Spring Batch cung cấp 1 Tasklet implement cho việc gọi system commands:

|  |
| --- |
| <bean class="org.springframework.batch.core.step.tasklet.SystemCommandTasklet">  <property name="command" value="echo hello" />  <!-- 5 second timeout for the command to complete -->  <property name="timeout" value="5000" />  </bean> |

## Handling Step Completion When No Input is Found

- Sử dụng listener được cung cấp sau:

|  |
| --- |
| public class NoWorkFoundStepExecutionListener extends StepExecutionListenerSupport {  public ExitStatus afterStep(StepExecution stepExecution) {  if (stepExecution.getReadCount() == 0) {  return ExitStatus.FAILED; //the Step sẽ fail  }  return null; // sẽ không ảnh hưởng đến status của Step.  }  } |

## Passing Data to Future Steps

- Thường được dùng để truyền information từ 1 Step đến 1 Step khác. Có thể dùng *ExecutionContext*. Có 2 *ExecutionContext*: 1 tại level Step và 1 tại level Job. Step *ExecutionContext* chỉ sống lâu (live as long as) ở trong Step, được cập nhật mỗi lần Step commits 1 chunk. Job *ExecutionContext* sống xuyên suốt cả Job, được cập nhật duy nhất tại the end của mỗi Step.

- Tất cả data phải được đặt trong Step *ExecutionContext* trong khi Step đang executing. Điều này chắc chắn data sẽ được lưu trong khi Step đang on-going. Nếu data được lưu trong Job *ExecutionContext*, thì nó sẽ không được tạo trong suốt Step execution và nếu Step fail, thì data sẽ bị mất.

|  |
| --- |
| public class SavingItemWriter implements ItemWriter<Object> {  private StepExecution stepExecution;  public void write(List<? extends Object> items) throws Exception {  // ...  ExecutionContext stepContext = this.stepExecution.getExecutionContext();  stepContext.put("someKey", someObject);  }  @BeforeStep  public void saveStepExecution(StepExecution stepExecution) {  this.stepExecution = stepExecution;  }  } |

- Để làm cho data hiển thị cho future Steps, thì nó phải được ‘promoted’ cho Job *ExecutionContext* sau khi Step đã finished. Spring Batch cung cấp *ExecutionContextPromotionListener*. Listener phải được cấu hình với keys có quan hệ với data(phải được promoted) trong *ExecutionContext*. Giống như các listener khác, phải register trong Step.

|  |
| --- |
| <job id="job1">  <step id="step1">  <tasklet>  <chunk reader="reader" writer="savingWriter" commit-interval="10"/>  </tasklet>  **<listeners>**  **<listener ref="promotionListener"/>**  **</listeners>**  </step>  <step id="step2">  ...  </step>  </job>  **<beans:bean id="promotionListener" class="org.spr....ExecutionContextPromotionListener">**  **<beans:property name="keys" value="someKey"/>**  **</beans:bean>** |

- Cuối cùng, những giá trị đã lưu phải được khôi phục từ Job *ExecutionContext*:

|  |
| --- |
| public class RetrievingItemWriter implements ItemWriter<Object> {  private Object someObject;  public void write(List<? extends Object> items) throws Exception {  // ...  }  @BeforeStep  public void retrieveInterstepData(StepExecution stepExecution) {  JobExecution jobExecution = stepExecution.getJobExecution();  ExecutionContext jobContext = jobExecution.getExecutionContext();  this.someObject = jobContext.get("someKey");  }  } |

# Unit Testing

* Theo thứ tự cho việc chạy unit test cho bactch job, farme work phải load applicationContext của Job. Có hai annotation được dùng là:
* @RunWith (SpringJUnit4ClassRunner.class): class khi dùng annotation này sẽ dùng JUnit của Spring để test.
* @ContextConfiguration(location = {...}): xác định file xml nào được sử dụn trong ApplicationContext.
* Ví dụ:

|  |
| --- |
| *@RunWith(SpringJUnit4ClassRunner.class)* *@ContextConfiguration(locations = { "/simple-job-launcher-context.xml",  "/jobs/skipSampleJob.xml" })* **public** **class** SkipSampleFunctionalTests { ... } |

## End-To-End Testing of Batch Jobs:

* ‘End To End’ tesing có thể định nghĩa giôgsn như testing việc hoàn thành run của bach job từ lúc bắt đầu đến lúc kết thúc. Điều đó cung cấp cho test có khả năng thiết lập một test với condition, execute job và verifies kết quả cuối cùng.
* Trong ví dụ dưới đây, batch job đọc từ database và writes đến flat file. Phương thức test bắt đầu bằng việt thiết lập database với test data. Nó xoá dữ liệu trong bảng Customer và thêm vào đó 10 records. Tiếp theo Test chạy Job bằng cách sử dụng phương thức lauchJob. Phương thức launchJob được cung cấp bởi JobLauhcherTestUtils, cũng cung cấp bởi class dùng chung là launchJob(JobParameters) nơi mà cho phép việc test các parameter cụ thể. Phương thức launchJob() trả về JobExecution, nó hữu ích cho việc assert thông tin về Job run. Trong trường hợp bên dưới, Test sẻ verifies Job end vs statsu là “COMPLETED”

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.class)  @ContextConfiguration(locations = { "/simple-job-launcher-context.xml",  "/jobs/skipSampleJob.xml" })  public class SkipSampleFunctionalTests {  @Autowired  private JobLauncherTestUtils jobLauncherTestUtils;  private SimpleJdbcTemplate simpleJdbcTemplate;  @Autowired  public void setDataSource(DataSource dataSource) {  this.simpleJdbcTemplate = new SimpleJdbcTemplate(dataSource);  }  @Test  public void testJob() throws Exception {  simpleJdbcTemplate.update("delete from CUSTOMER");  for (int i = 1; i <= 10; i++) {  simpleJdbcTemplate.update("insert into CUSTOMER values (?, 0, ?, 100000)",  i, "customer" + i);  }  JobExecution jobExecution = jobLauncherTestUtils.launchJob().getStatus();  Assert.assertEquals("COMPLETED", jobExecution.getExitStatus());  }  } |

## Testing Individual Steps:

* Đối với các batch job phức tạp, test case trong việc tiêp cận của end-to-end testing có thể không thể quản lý. Trong trường hợp này, nó có thể có nhiều hữu ích để có test case để test từng step riêng lẻ.
* AbstractJobTests class chứa phương thức launchStep để lấy tên của step và run cụ thể vs tên đó. Việc tiếp cận như thế này cho phép việc mở rông nhiều mục địch test bằng việc cho phép test có thể thiết lập data cho chỉ step và validate kết quả đó trực tiếp.

|  |
| --- |
| JobExecution jobExecution = jobLauncherTestUtils.launchStep("loadFileStep"); |

## Testing Step-Scoped Components:

* Thường thì các components mà dùng để config cho step tại runtime dùng step là scope và sau liên kết đến context trong step hoặc job execution.Sẽ tốt hơn nếu test các components riêng lẻ cho đến khi bạn có cách set context giống như nó đang trong step execution.
* Với mục đích trên, spring batch cung cấp 2 lớp StepScopeTestExecutionListenser và StepScopeTestUtils.
* Listener được khai báo trong class level và job của nó là tạo step execution step cho mỗi phương thức test. Ví dụ

|  |
| --- |
| @ContextConfiguration  @TestExecutionListeners( { DependencyInjectionTestExecutionListener.class,  StepScopeTestExecutionListener.class })  @RunWith(SpringJUnit4ClassRunner.class)  public class StepScopeTestExecutionListenerIntegrationTests {  // This component is defined step-scoped, so it cannot be injected unless  // a step is active...  @Autowired  private ItemReader<String> reader;  public StepExecution getStepExection() {  StepExecution execution = MetaDataInstanceFactory.createStepExecution();  execution.getExecutionContext().putString("input.data", "foo,bar,spam");  return execution;  }  @Test  public void testReader() {  // The reader is initialized and bound to the input data  assertNotNull(reader.read());  }  } |

* Có hai TestExecutionListeners, một từ SpringTest framework và handler sự phụ thuộc việc truyền dữ liệu từ config của application context, truyền dữ liệu reader và một cái khác là Spring Batch StepScopeTestExecutionListerner. Nó làm việcbằng cách nhìn cho phương thức factory trong test case cho StepExecution và sử dùng điều đó giôgns như context của phương thức test, giống như nếu execution được active trong step ở giai đoạn runtime. Phương thức factory được phát hiện bằng chữ ký (nó chỉ phải trả về StepExecution). Nếu phương thức factory không cung cấp thì tiếp túc StepExecution mặc định sẽ được tạo.
* Phương pháo tiếp cận của listener là tiện lợi nếu người dùng muốn trong khi chạy step scope trở thành execution của phương thức test. Để linh hoạt hơn, nhưng muốn một phương pháp có khả năng xấm lấn nhiều hơn, người dùng có thể dùng StepScopeTestUtils. Ví dụ, để đếm số lượng items hiện hữu trong reader ở trên:

|  |
| --- |
| int count = StepScopeTestUtils.doInStepScope(stepExecution,  new Callable<Integer>() {  public Integer call() throws Exception {  int count = 0;  while (reader.read() != null) {  count++;  }  return count;  }  }); |

## Validating Output files:

* Khi batch job write đến database, nó rất dễ dàng để truy vấn database để verify output được kì vọng.
* Tuy nhiên, nếu batch job writes đến file, nó cũng quan trọn không kém là output được verified. Spring batch cung cấp AssertFile để thuận tiện để verify output file.
* Phương thức assertFileEquals lấy hai file objects và asserts, line by line, điều đó có nghĩa là 2 file phải có chung context. Vì thế, nó có thể tạo file với kì vọng output và so sánh với kết quả thực tế.

|  |
| --- |
| private static final String EXPECTED\_FILE = "src/main/resources/data/input.txt";  private static final String OUTPUT\_FILE = "target/test-outputs/output.txt";  AssertFile.assertFileEquals(new FileSystemResource(EXPECTED\_FILE),  new FileSystemResource(OUTPUT\_FILE)); |

## Mocking Domain Objects:

* Một issue thường gặp là trong khi unit và integration test của compoents trong Spring batch là làm thế nào để mock domain objects. Một ví dụ là StepExecutionListener như dưới đây:

|  |
| --- |
| public class NoWorkFoundStepExecutionListener extends StepExecutionListenerSupport {  public ExitStatus afterStep(StepExecution stepExecution) {  if (stepExecution.getReadCount() == 0) {  throw new NoWorkFoundException("Step has not processed any items");  }  return stepExecution.getExitStatus();  }  } |

* Listener phía trên được cung cấp bởi framework và kiểm tra StepExecution cho một read count trông, do đó biểu thị là không của việc nào done. Ví dụ này là khá đơn giản, nó mô tả một loại problems mà có thể bắt gặp khi thực hiện attempt để unit test class mà cần phải implement interface của domain object:

|  |
| --- |
| private NoWorkFoundStepExecutionListener tested = new NoWorkFoundStepExecutionListener();  @Test  public void testAfterStep() {  StepExecution stepExecution = new StepExecution("NoProcessingStep",  new JobExecution(new JobInstance(1L, new JobParameters(),  "NoProcessingJob")));  stepExecution.setReadCount(0);  try {  tested.afterStep(stepExecution);  fail();  } catch (NoWorkFoundException e) {  assertEquals("Step has not processed any items", e.getMessage());  }  } |

* Bởi vì Spring batch domain model theo cấu trúc nguyên tắc hương đối tượng, StepExecution yêu cầu có JobExeution, cái mà yêu cầu JobInstance và JobParmeters trong lệnh để valid StepExecution. Trong khi nó tốt trong mô hình domain model, nó tạo ra objects sơ khai cho unit testing. Để có địa chỉ đến issue, Spring batch test module bao gồm factory của creating domain object: MetaDataInstanceFactoty. Cho facetory này, unit test có thể update để ngắn gọn hơn:

|  |
| --- |
| private NoWorkFoundStepExecutionListener tested = new NoWorkFoundStepExecutionListener();  @Test  public void testAfterStep() {  StepExecution stepExecution = MetaDataInstanceFactory.createStepExecution();  stepExecution.setReadCount(0);  try {  tested.afterStep(stepExecution);  fail();  } catch (NoWorkFoundException e) {  assertEquals("Step has not processed any items", e.getMessage());  }  } |

* Phương thức nêu trên tạo ra simple StepExecution chỉ có một phương thức tiên ích hiện hữu trong factory.