**STEPS GESTIONE STATO:  
  
PROBLEMA:**

**Chunk Oriented:**Condividere dati tra uno step (Read-Write) CO ed il successivo.

**Tasklet:**Salva tra uno step Tasklet ed il successivo il numero generato.

**DEMO:**

1. Generatore di numeri casuali
2. Step1:
   1. genera N numeri sequenziali (read fino a contatore)
   2. li stampa tutti insieme (in transazione)
   3. salva il suo **ultimo numero** processato
3. Step2:
   1. recupera **ultimo numero** di Step1
   2. Riprende operazione da ultimo numero sommandogli quelli nuovi.

**IMPL:**

NumberGenerator:  
1) (prototype) ogni Reader ha la sua copia   
2) Genera numeri da 1 a 100.

**TEST:**

Se tutto va bene   
il primo step stampera numeri da 1 a 100 in transazioni da 10.  
Il secondo step da 101 a 200.

**SOLUZIONI: (TEORIA)**

**JobContext:**

Ci sono due contesti dove salvare i dati accessibili dallo StepExecution:  
  
1) Livello Step:  
Update: ogni volta che viene committato un chunk in uno step  
LIfeCycle: condiviso tra n esecuzioni dello stesso Step (non condivisibile tra step, ma fra le eventuali ripetizioni step)

2) Livello Job:  
Update: ogni volta che viene committato un chunk in uno step  
LifeCycle: Ripulito alla fine dello Step (non condivisibile)  
  
PRO: Condiviso eventualmente tra N server. Salvato su DB  
CONTRO: Problemi per dati di grandi dimensioni.

**BeanHolder: (semplice, non necessita Spring)**

Creo un singleton bean con Spring contenente le collections necessarie alla condivisione.  
  
PRO: utile per dati di grandi dimensioni che altrimenti verrebbero salvati su jobrepository.  
CONTRO: Soluzione meno scalabile di Context.

**STEPS:**

**CHUNK ORIENTED:**

1. Nei Reader/Writeri dello Step accesso allo StepExecution
2. Da questo recuperare
   1. STEP: lo StepContext e put/get chiavi valori
   2. JOB: Il JobContext e put/get chiavi valori
3. Tests:
   1. Step e Job Level
      1. Dati salvati su DB con le due opzioni
      2. A livello Step il Reader Step2 non riceve dati con JobLevel si.
   2. A che Serve Step Level:
      1. Provare a far fallire il Writer@S1 e verificare che I dati a livello Step ci sono ancora (ripetere da li)

**Dettagli ChunkOriented Impl:**

1. Accesso a oggetto stepExecution (contesto Step) in classi Step:

@BeforeStep

**public** **void** saveStepExecution(StepExecution stepExecution) {

**this**.stepExecution = stepExecution;

}

1. Accesso a context livello **Step**:  
     
   ExecutionContext stepExecutionCtx = stepExecution.getExecutionContext();
2. Accesso a context livello **Job**:

JobExecution jobExecution = stepExecution.getJobExecution();

ExecutionContext jobExecutionCtx=jobExecution.getExecutionContext();

**Dettagli TASKLETS Impl:**

1. Accesso a oggetto stepExecution (contesto Step) in classi Step:

@Override

**public** RepeatStatus execute(StepContribution stepContribution, **ChunkContext** chunkContext) **throws** Exception {

1. Accesso a context livello **Step**:

chunkContext.getStepContext().getStepExecutionContext();

1. Accesso a context livello **Job**:

StepExecution stepExecution=chunkContext.getStepContext().getStepExecution();

JobExecution jobExecution = stepExecution.getJobExecution();

ExecutionContext jobExecutionCtx=jobExecution.getExecutionContext();

jobExecutionCtx.put(Consts.***VALUE\_KEY, value***);

**N.B: Sembra non essere necessaria la promozione a Job Context del parametro.  
  
REFERENCE:**

**11.8 Passing Data to Future Steps**

It is often useful to pass information from one step to another. This can be done using  
the ExecutionContext.   
The catch is that there are two ExecutionContexts: one at the Step level and one at the Job level.   
The Step ExecutionContext lives only as long as the step while the Job ExecutionContext lives through the whole Job.   
  
On the other hand, the Step ExecutionContext is updated every time the Step commits a chunk while the Job ExecutionContext is updated only at the end of each Step.

The consequence of this separation is that all data must be placed in the Step ExecutionContext while the Step is executing.   
This will ensure that the data will be stored properly while the Step is on-going. If data is stored to the Job ExecutionContext, then it will not be persisted  
during Step execution and if the Step fails, that data will be lost.

**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;

}

}

To make the data available to future Steps, it will have to be "promoted" to the Job ExecutionContext after the step has finished. Spring Batch provides theExecutionContextPromotionListener for this purpose. The listener must be configured with the keys related to the data in the ExecutionContext that must be promoted. It can also, optionally, be configured with a list of exit code patterns for which the promotion should occur ("COMPLETED" is the default). As with all listeners, it must be registered on the 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>**

Finally, the saved values must be retrieved from the Job ExeuctionContext:

**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");

}

}