**DEMO CHUNK ORIENTED NOTE:**

Demo Bonifici Interbancari:  
BonificoDto.

1. Trasformazioni
   1. Panoramica COP classi esistenti
2. Validazioni
   1. Processors
3. Filtri
4. Aggregazione
5. Split

JDBC Cursor Processing

execute a query by opening a cursor to return results on demand. To do that, you will use Spring Batch’s

org.springframework.batch.item.database.JdbcCursorItemReader. This ItemReader opens a cursor (by

creating a ResultSet) and have a row mapped to a domain object each time the read method is called by

Spring Batch.

JDBC Paged Processing

When working with a paginated approach, Spring Batch returns the result set in chunks called pages.

Each page is a predefined number of records to be returned by the database. It is important to note that

when working with pages, the items your job will process will still be processed individually. There is no

difference in the processing of the records. What differs is the way they are retrieved from the database.

Instead of retrieving records one at a time, paging will essentially cache a page until they are needed to

be processed.

**Impl Paging:**

For example, if your total

number of records is 10,000 and your page size is 100 records, you need to be able to specify that you are

requesting the 20th page of 100 records (or records 2,000 through 2100). To do this, you provide an

implementation of the org.springframework.batch.item.database.PagingQueryProvider interface to the

JdbcPagingItemReader. The PagingQueryProvider interface provides all of the functionality required to

navigate a paged ResultSet.

Unfortunately, each database offers its own paging implementation. Because of this, you have the

following two options:

1. Configure a database-specific implementation of the PagingQueryProvider. As

of this writing, Spring Batch provides implementations for DB2, Derby, H2,

HSql, MySql, Oracle, Postgres, SqlServer, and Sybase.

2. Configure your reader to use the

org.springframework.batch.item.database.support.SqlPagingQueryProviderF

actoryBean. This factory detects what database implementation to use.

Although the easier route is definitely the SqlPagingQueryProviderFactoryBean, it is important to

note that each of the different databases implement paging in a different way. Because of this, you may

want to use database specific options when tuning your jobs.

<beans:property name="dataSource" ref="dataSource"/>

<beans:property name="queryProvider">

<beans:bean class="org.springframework.batch.item.database.support.

SqlPagingQueryProviderFactoryBean">

<beans:property name="selectClause" value="select \*"/>

<beans:property name="fromClause" value="from Customer"/>

<beans:property name="whereClause" value="where city = :city"/>

<beans:property name="sortKey" value="lastName"/>

<beans:property name="dataSource" ref="dataSource"/>

</beans:bean>

</beans:property>

RowMapper implementation that will be used to map your results.

Within the PagingQueryProvider’s configuration, you provide five pieces of information. The first

three are the different pieces of your SQL statement: the select clause, the from clause, and the where

clause of your statement. The next property you set is the sort key. It is important to sort your results

when paging since instead of a single query being executed and the results being streamed, a paged

approach will typically execute a query for each page. In order for the record order to be guaranteed

across query executions, an order by is recommended and is applied to the generated SQL statement for

any fields that are listed in the sortKey. Finally, you have a dataSource reference. You may wonder why

you need to configure it in both the SqlPagingQueryProviderFactoryBean and the

JdbcPagingItemReader. The SqlPagingQueryProviderFactoryBean uses the dataSource to determine

what type of database it’s working with. From there, it provides the appropriate implementation of the

PagingQueryProvider to be used for your reader.

**HIBERNATE:**

For batch processing, if you use Hibernate naively, you would use the normal stateful session

implementation, read from it as you process your items, and write to it as you complete your processing

closing the session once the step is complete. However, as mentioned, the standard session within Hibernate is stateful. If you are reading a million items, processing them, then writing those same

million items, the Hibernate session will cache the items as they are read and an

OutOfMemoryException will occur.

Another issue with using Hibernate as a persistence framework for batch processing is that

Hibernate incurs larger overhead than straight JDBC does. When processing millions of records, every millisecond can make a big difference.

**Jpa:**

JPA does not

support cursor database access but it does support paging database access. The ItemReader will be the

org.springframework.batch.item.database.JpaPagingItemReader.

**ITEM READER ADAPTER:**

To use an existing service within Spring Batch, the same pattern is used.

In this case, you will be using the org.springframework.batch.item.adapter.ItemReaderAdapter.

This class takes two dependencies when it is configured: a reference to the service to call and the name of the method to call.   
  
You need to keep the following two things in mind when using the ItemReaderAdapter:

1. The object returned from each call is the object that will be returned by the

ItemReader. If your service returns a single Customer, then that single

Customer object will be the object passed onto the ItemProcessor and finally

the ItemWriter. If a collection of Customer objects is returned by the service, it

will be passed as a single item to the ItemProcessor and ItemWriter and it will

be your responsibility to iterate over the collection.

2. Once the input is exhausted, the service method must return a null. This

indicates to Spring Batch that the input is exhausted for this step.

**test:**  
For this example, you will use a service hardcoded to return a Customer object for each call until the list is exhausted. Once the List is exhausted, null will be returned for every call after

**Implementare un ItemReader stateful**Per evitare ogni volta di ripartire daccapo, ma invece mantenere lo stato tra le esecuzioni.  
  
Implementare ANCHE interfaccia ItemStream

The ItemStream Interface

package org.springframework.batch.item;

public interface ItemStream {

void open(ExecutionContext executionContext) throws ItemStreamException;

void update(ExecutionContext executionContext) throws ItemStreamException;

void close() throws ItemStreamException;

}

execution of a step.   
open is called to initialize any required state within your ItemReader.   
This includes the opening of any files or database connections as well as when restarting a job. The open method could be used to reload the number of records that had been processed so they could be skipped during the second execution.   
  
update is used by Spring Batch as processing occurs to update that state.   
Keeping track of how many records or chunks have been processed is a use for the update method.   
  
Finally, the close method is used to close any required resources (close files, etc).

You will notice that the open and update provide access to the ExecutionContext that you did not have a handle on in your ItemReader implementation.   
This is because Spring Batch will use the open method to reset the state of the reader when a job is restarted. It will also use the update method to learn the current state of the reader (which record you are currently on) as each item is processed.   
Finally, the close method is used to clean up any resources used in the ItemStream.

Now you may be wondering how you can use the ItemStream interface for your ItemReader if it

doesn’t have the read method. Short answer: you don’t. Instead you’ll use a utility interface,

**org.springframework.batch.item.ItemStreamReader**, that **extends both the ItemStream and the ItemReader interfaces.**