**READERS:**

**ItemReader**

package org.springframework.batch.item;

public interface ItemReader<T> {

T read() throws Exception, UnexpectedInputException, ParseException,

NonTransientResourceException;

}

**FLAT FILES:**

**XML PARSING:**

1. resource
2. marshaller (Obj 2 Xml e viceversa)
3. org.springframework.batch.item.xml.StaxEventItemReader (Stax reader logic)

**MARSHALLER:**

To parse your customer file, there are three things you will need to configure on the XStreamMarshaller instance.

1. *Aliases*: This is a map of tag names to fully qualified class names that tells the

unmarshaller what each tag maps to.

2. *implicitCollection*: This is a map of fields to fully qualified classes that indicate

what fields on the class specified are collections consisting of another type.

3. *Converters*: Although XStream is pretty smart and can figure out how to

convert most of your XML file from the Strings it sees in the file to the required

data type in your objects, you need to help it out on the transaction date.   
  
For XStream to be able to parse the transaction date, you will need to provide a DateConverter instance configured with the correct date format.

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.

Ha bisogno di :

1)

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.

1. The first three are the different pieces of your SQL statement: the select clause, the from clause, and the where clause of your statement.
2. 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.
3. 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.

**DELEGA A SERVIZI CON ADAPTERS:**

Rispettare criteri ItemReader, ItemProcessor, ItemWriter:  
  
ItemReader restituisce null quando ha esaurito il dataset in lettura  
  
Il tipo restituito dal Reader deve essere l input del Processor

Il tipo restituito dal Processor deve essere input di Writer (singolo)  
  
Esecuzione:  
1) Tutti i read del commit interval  
2) Tutti i process del commit interval  
3) Tutti i write del commit interval

**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

**CUSTOM ITEM READER (DA aggiungere alla demo Status)**

**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.**

**PROCESSORS:**

From Minnella:

*Validate input*: In the original version of Spring Batch, validation occurred at the

ItemReader by subclassing the ValidatingItemReader class. The issue with this

approach is that none of the provided readers subclassed the

ValidatingItemReader class so if you wanted validation, you couldn’t use any of

the included readers.   
  
Moving the validation step to the ItemProcessor allows

validation to occur on an object before processing, regardless of the input method.

This makes much more sense from a division-of-concerns perspective.

*Reuse existing services*: Just like the ItemReaderAdapter you looked at in Chapter 7 to reuse services for your input, Spring Batch provides an ItemProcessorAdapter for the same reason.

*Chain ItemProcessors*: There are situations where you will want to perform

multiple actions on a single item within the same transaction. Although you could

write your own custom ItemProcessor to do all of the logic in a single class, that

couples your logic to the framework, which is something you want to avoid.

Instead, Spring Batch allows you to create a list of ItemProcessors that will be

executed in order against each item.

1. CompositeItemProcessor
   1. Leggi tabelle specifiche
   2. Sulla base dei dati letti fai un calcolo
   3. Risultato usato da prossimo Processor nella chain

**WRITERS:**

**FlatFileItemWriter:**

**Si configura con**

1. Resource
2. LineAggregator

Implementazioni di LineAggregator

replaces the LineMapper of the FlatFileItemReader discussed in Chapter 7. Here, instead of parsing a String into an object as the LineMapper is responsible for doing, the LineAggregator is responsible for the generating of an output String based on an object.

In many of the previous examples, you used **PassThroughLineAggregator** to generate the output files.

This particular implementation of the LineAggregator interface just calls the item’s toString() method to generate the output.

Altre implementazioni di LineAggregator

1. FormatterLineAggregator:

Formatta la riga di output basandosi su un template (**format**) nel quale i %s vengono sostituiti dalle props del bean elencate nel **fieldExtractor**

<beans:bean id="formattedLineAggregator"

class="org.springframework.batch.item.file.transform.

FormatterLineAggregator">

<beans:property name="**fieldExtractor**">

<beans:bean class="org.springframework.batch.item.file.transform.

**BeanWrapperFieldExtractor**">

<beans:property name="names"

value="firstName,lastName,address,city,state,zip"/>

</beans:bean>

</beans:property>

<beans:property name="**format**" value="%s %s lives at %s %s in %s, %s."/>

</beans:bean>

1. DelimitedLineAggregator:

Usato per output files dove ogni riga ha un numero di tokens (teoricamente) diverso.  
  
Da configurare  
2.1) FieldExtractor (come prima) che elenca l ordine con cui vengono recuperati i campi

2.2) delimiter

<beans:bean id="delimitedLineAggregator" class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">

<beans:property name="fieldExtractor">

<beans:bean class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor">

<beans:property name="names" value="zip,state,city,address,lastName,firstName"/>

</beans:bean>

</beans:property>

<beans:property name="delimiter" value=";"/>

</beans:bean>

**Gestione files (a fine step)**1) Append, CreateIfMissing, DeleteIfEmpty

**JDBC JdbcBatchItemWriter**

JdbcBatchItemWriter uses the JdbcTemplate and its batch SQL execution capabilities to execute all of the SQL for a single chunk at once.

org.springframework.batch.item.database.JdbcBatchItemWriter isn’t much more than a thin

wrapper around Spring’s org.springframework.jdbc.support.JdbcTemplate, using the

**JdbcTemplate.batchUpdate or JdbcTemplate.execute** method depending on if named parameters are

used in the SQL to execute mass database insert/updates.   
  
The important thing to note about this is that

Spring uses PreparedStatement’s batch-update capabilities to execute all the SQL statements for a single

chunk at once instead of using multiple calls. This greatly improves performance while still allowing all

the executions to execute within the current transaction.

**Named Parameters invece che PreparedStatement**

named parameter approach provided by Spring’s JdbcTemplate is a much safer way to go and is the

preferred way to populate parameters in most Spring environments. With that in mind, you can put this

feature to use by making two small updates to the configuration:

1. Update the configuration to remove the ItemPreparedStatementSetter

implementation you wrote and replace it with an implementation of the

ItemSqlParameterSourceProvider interface.

2. Update the SQL to use named parameters instead of question marks for

parameters.

The org.springframework.batch.item.database.ItemSqlParameterSourceProvider interface is

slightly different from the ItemPreparedStatementSetter interface in that it doesn’t set the parameters on the statement to be executed.   
  
Instead, an **implementation of the ItemSqlParameterSourceProvider’s** responsibility is to extract the parameter values from an item and return them as an org.springframework.jdbc.core.namedparam.SqlParameterSource object.

The nice thing about this approach is that not only is it the safer approach (no concerns about

needing to keep the SQL in the XML file in synch with the code of the ItemPreparedStatementSetter implementation) but Spring Batch provides implementations of this interface that allow you to use convention over code to extract the values from the items.

In this example, you use Spring Batch’s

BeanPropertyItemSqlParameterSourceProvider (try saying that three times fast) to extract the values from the items to be populated in the SQL.  
Listing 9-30 shows the updated jdbcBatchWriter configuration for this change.

**PropertyExtractingDelegatingItemWriter**

The use case for ItemWriterAdapter is pretty simple.

Take the item being processed, and pass it to an existing Spring service. However, software is rarely that straightforward. Because of that, Spring Batch has provided a mechanism to extract values from an item and pass them to a service as parameters.   
  
This section looks at PropertyExtractingDelegatingItemWriter and how to use it with an existing service.

Although it has a long name,

org.springframework.batch.item.adapter.PropertyExtractingDelegatingItemWriter is a lot like the

ItemWriterAdapter.   
  
Just like ItemWriterAdapter, it calls a specified method on a referenced Spring service.   
The difference is that instead of blindly passing the item being processed by the step,

PropertyExtractingDelegatingItemWriter passes only the attributes of the item that are requested.   
  
For example, if you have an item of type Product that contains fields for a database id, name, price, and SKU number, you’re required to pass the entire Product object to the service method as with

ItemWriterAdapter.   
But with PropertyExtractingDelegatingItemWriter, you can specify that you only want the database id and price to be passed as parameters to the service

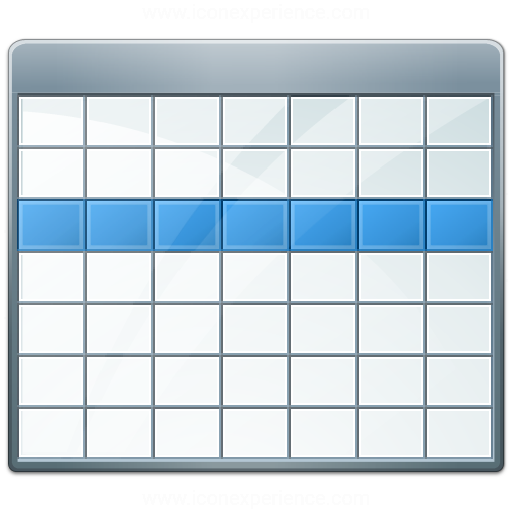
**ˇ**

**JmsItemWriter**

**SimpleMailMessageItemWriter**

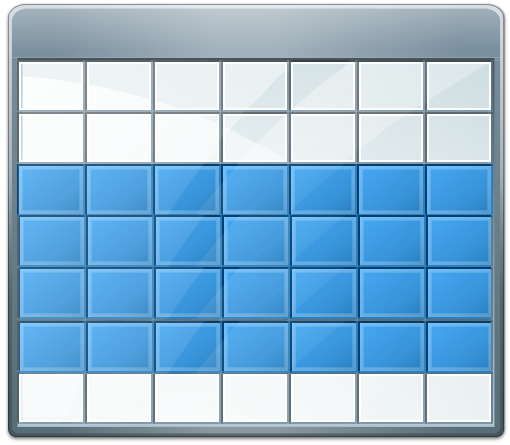
**MultiResourceItemWriter**

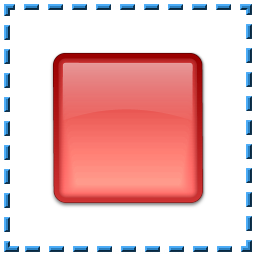
**ICONE:**



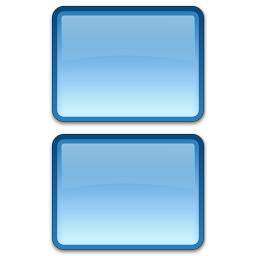
**OUTPUT**

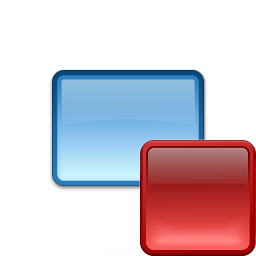


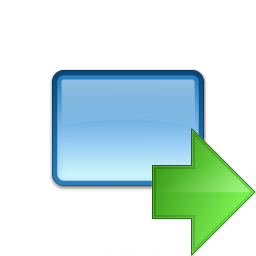
****

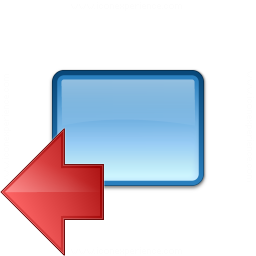






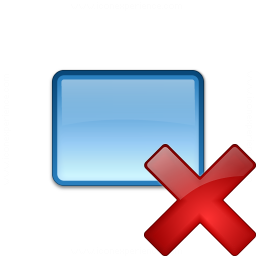


















[**https://www.iconexperience.com/v\_collection/icons/?icon=nav\_plain\_red**](https://www.iconexperience.com/v_collection/icons/?icon=nav_plain_red)















