Desmistificando o *"Spring Batch"*

Vanilson Burégio &

Walter Wanderley

Agenda

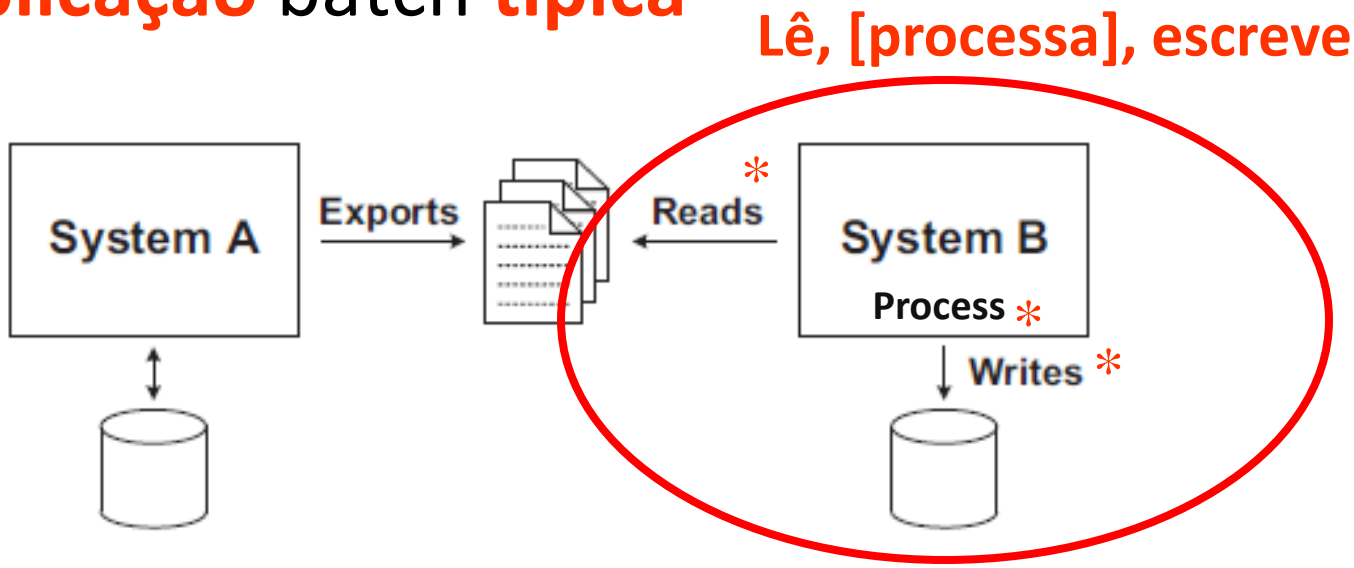
- Fundamentos do desenvolvimento de aplicações de processamento em lote;
- Introdução ao *Spring Batch*;
- Principais características e componentes;
- Tipos de passos;
  - Tasklet e processamento orientado a chunks;
- Leitura de dados
- Escrita de dados
- Controle de fluxo
- Uso de transações
  - Políticas de Skip e Retry;

Introdução

- O que é uma aplicação batch?

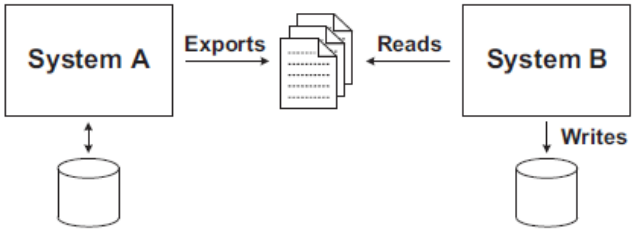
Introdução

- **Aplicação** batch **típica**



Sistema A **exporta** dados para um arquivo, e Sistema B usa um processamento batch para **ler** os arquivos e **inserir** em uma base de dados.

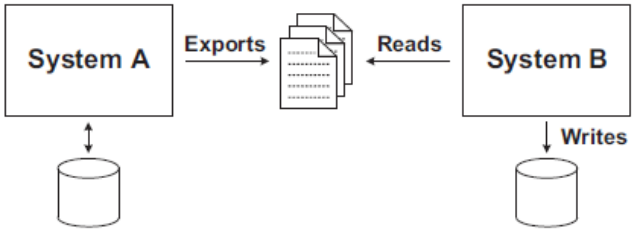
Introdução



Quais são os Requisitos?

- **Grande volume de dados**
  - Aplicações batch precisam ser capazes de manipular um grande volume de dados para importar, exportar ou computar
- **Automatização**
  - Aplicações batch devem executar sem interação com o usuário, com raríssimas exceções.
- **Robustez**
  - Aplicações batch devem lidar com dados inválidos sem falhar ou deixar de funcionar prematuramente.
- **Confiabilidade**
  - Aplicações batch devem manter o rastreamento de erros, registrando quando ocorreram (*logging*, notificação).

Introdução



Quais são os Requisitos?

➢ Desempenho

➢ Aplicações batch devem executar de forma performática, dentro de uma janela de tempo dedicada, evitando impactos em outras aplicações que rodam simultaneamente.

➢ Escalabilidade

➢ Capacidade de lidar com uma quantidade crescente de trabalho de forma satisfatória.

➢ Memória Vs Processamento

➢ Em geral, o tempo de processamento é o fator determinante da solução

Quais os requisitos/problemas do FIBRA com relação aos jobs de processamento em lote?

Introdução

- O que é o Spring Batch?

Primeiro framework Java para processamento batch

- Base para JSR 352
  - API do Java EE 7 para o desenvolvimento de processos em lote
- Leve, completo, produtivo
- Abordagem baseada em POJO - > Spring Framework

Spring Batch NÃO é um scheduler!

Introdução

- **Objetivos do Spring Batch**

➢ API para **facilitar a construção e parametrização de processos em lote** (**jobs**) através da definição de um fluxo de passos (***steps***)

➢ Passos são descritas em um **arquivo XML**

➢ cada job descrito em um arquivo XML separado (boa prática)

Spring Batch

- **Principais características**

Feature	Description
Spring Framework foundations	Benefits from enterprise support, dependency injection, and aspect-oriented programming
Batch-oriented processing	Enforces best practices when reading and writing data
Ready-to-use components	Provides components to address common batch scenarios (read and write data to and from databases and files)
Robustness and reliability	Allows for declarative skipping and retry; enables restart after failure

Spring Batch :: principais características

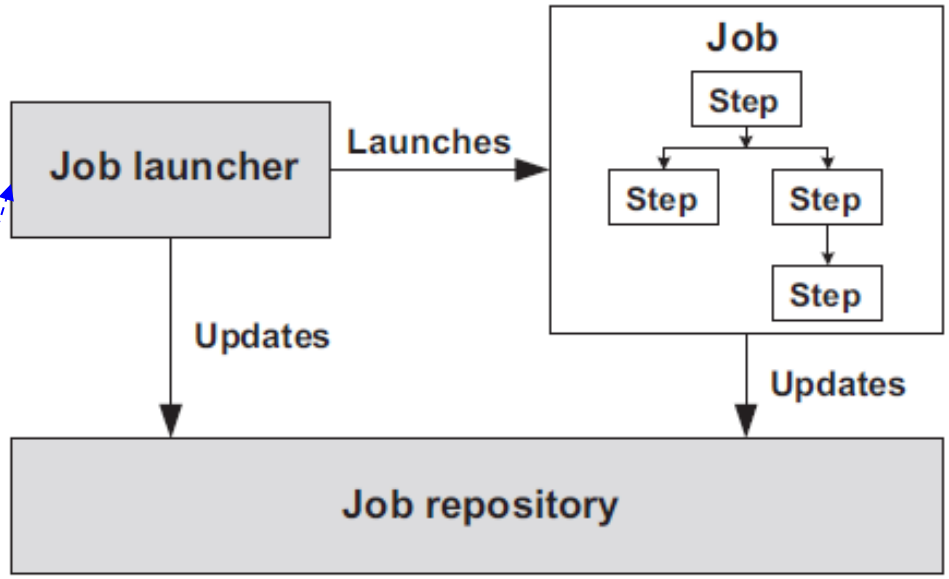
- Tecnologias de persistência suportadas

Data source type	Technology	Description
Database	JDBC	Leverages paging, cursors, and batch updates
Database	Hibernate	Leverages paging and cursors
Database	JPA (Java Persistence API)	Leverages paging
Database	iBATIS	Leverages paging
File	Flat file	Supports delimited and fixed-length flat files
File	XML	Uses StAX (Streaming API for XML) for parsing; builds on top of Spring OXM; supports JAXB (Java Architecture for XML Binding), XStream, and Castor

Spring Batch

- Conceitos básicos

Componentes de aplicação



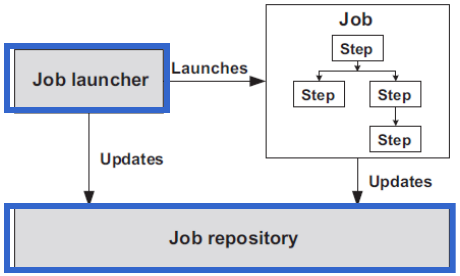
Desenvolvedor **configura** e **implementa** jobs

Componentes de infraestrutura

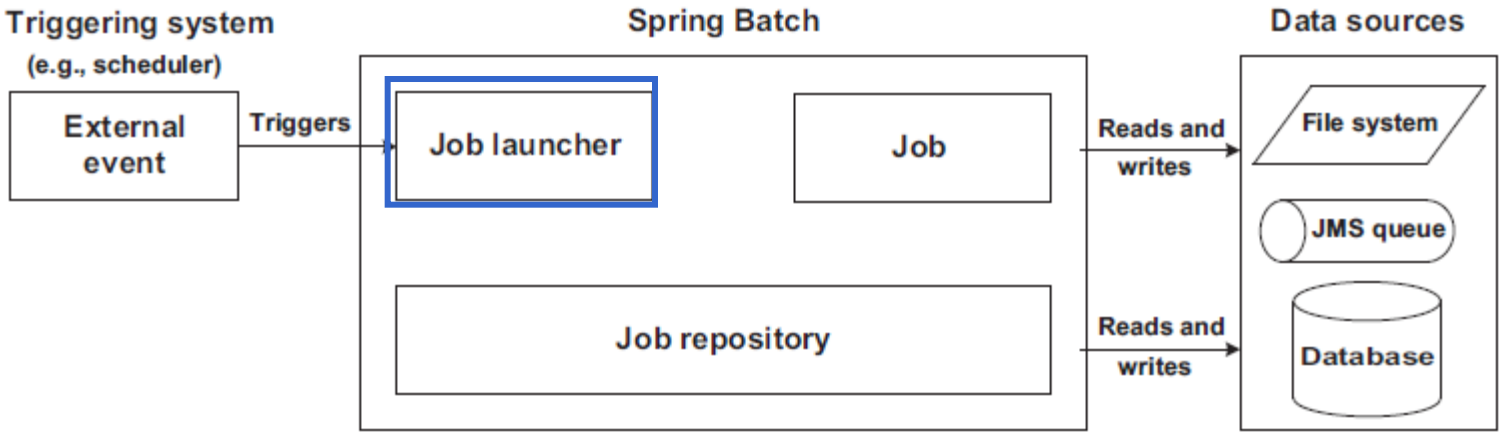
Job repository armazena metadados do job

Job launcher é utilizado para inicializar a execução de jobs

Spring Batch



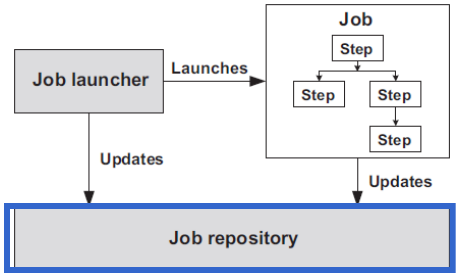
Componentes de infraestrutura

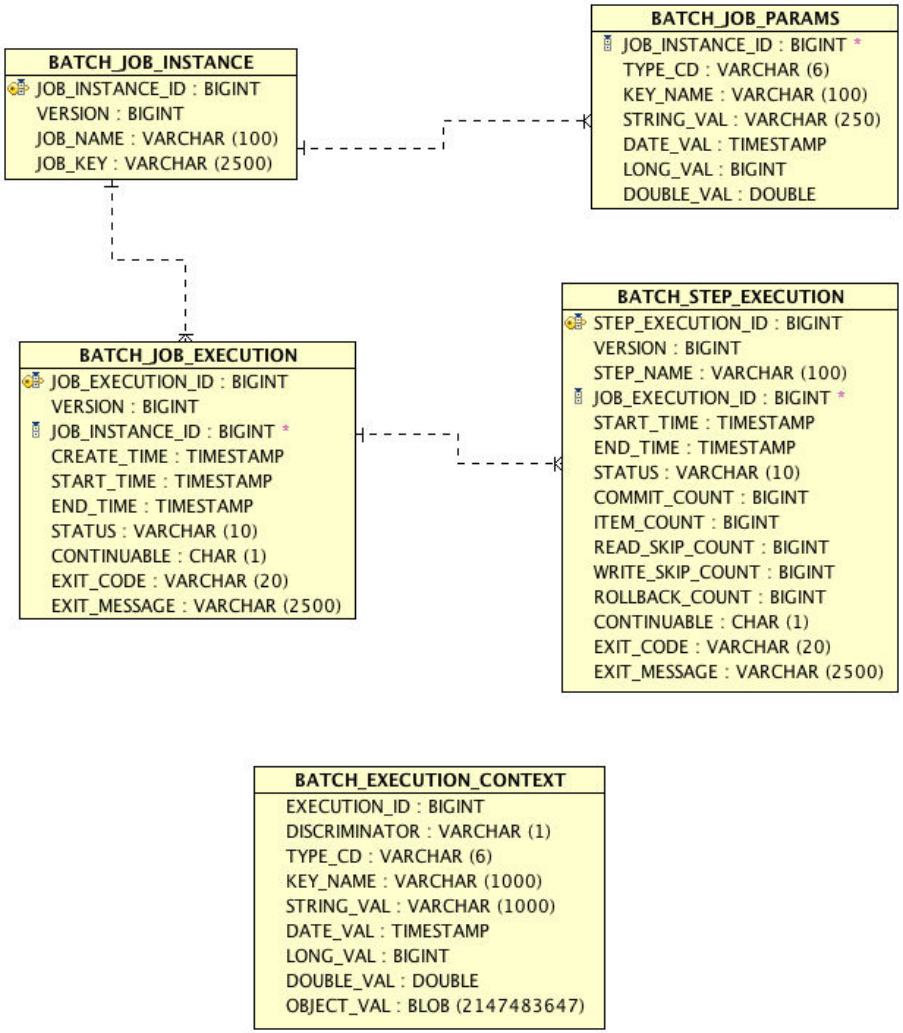



```
package org.springframework.batch.core.launch;

(...)

public interface JobLauncher {

 public JobExecution run(Job job, JobParameters jobParameters)
    throws JobExecutionAlreadyRunningException,
           JobRestartException, JobInstanceAlreadyCompleteException,
           JobParametersInvalidException;

}
```

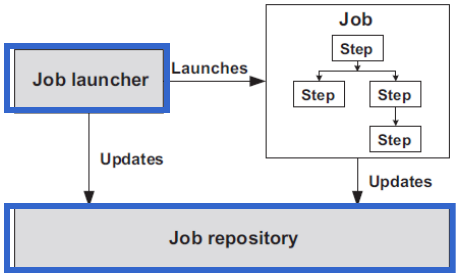
Spring Batch

- Job repository - Metadados





Spring Batch



Job repository

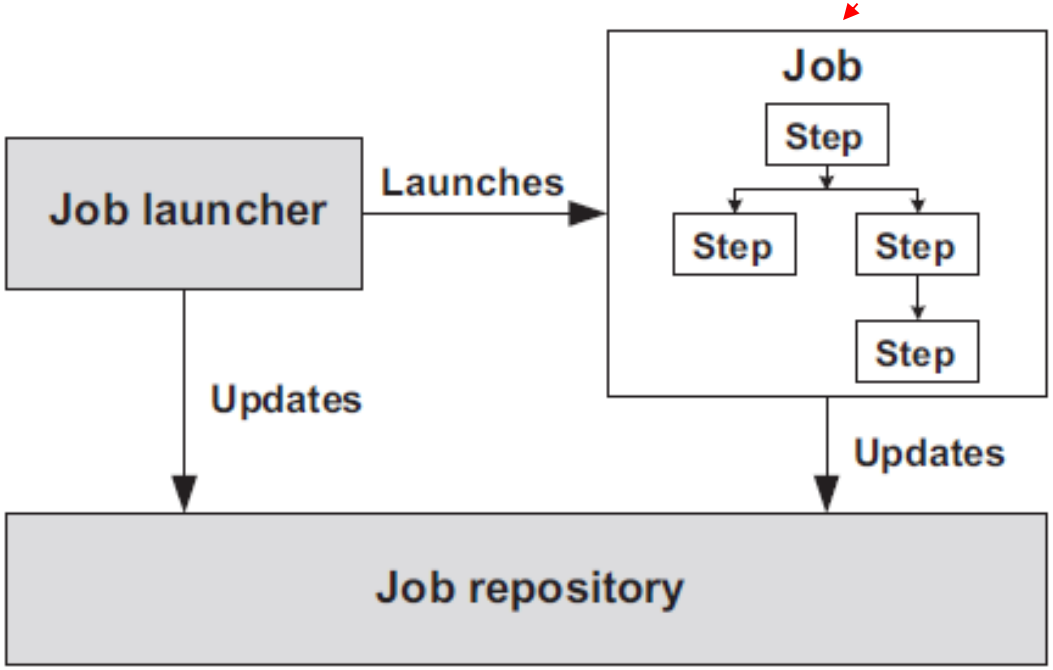
```
<batch:job-repository id="jobRepository"
  data-source="dataSource"
  transaction-manager="transactionManager" />
```

```
<bean id="jobLauncher"
 class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
  <property name="jobRepository" ref="jobRepository" />
</bean>
```

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.
SingleConnectionDataSource">
  <property name="driverClassName"
    value="org.h2.Driver" />
  <property name="url" value="
   jdbc:h2:mem:sbia_ch02;DB_CLOSE_DELAY=-1" />
  <property name="username" value="sa" />
  <property name="password" value="" />
  <property name="suppressClose" value="true" />
</bean>
```

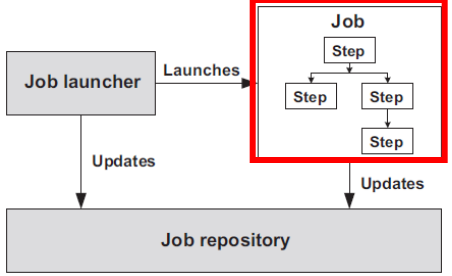
```
<bean id="transactionManager" class="org.springframework.jdbc.datasource.
  DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource" />
</bean>
```

Componentes de aplicação



Desenvolvedor **configura** e **implementa** jobs

Spring Batch



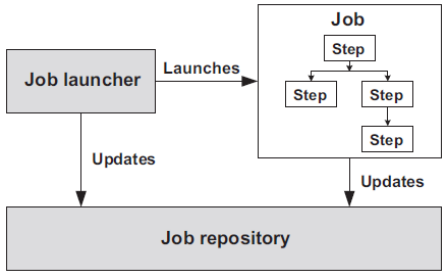
Aspectos de Desenvolvimento

✓Definição das cargas (***jobs***), passos (***steps***) e elementos de decisão (***decisions***) em nossas aplicações, sendo um processo batch representado por um *job*, que por sua vez consiste em um conjunto de *steps* inter-relacionados

✓Definição de um processo *batch* por meio de um fluxo completo de passos em um arquivo de configuração;

```
<!-- INICIO JOB ================================================================================================================== -->
<batch:job id="JOB-07509-CONSOLIDAR PCAD lab" restartable="false" parent="parentJob">

    <!-- #PASSO 00: Audita Triggers -->
    <batch:step id="CONSOLIDAR_PCAD-AUDITATRIGGERS" parent="parentStep" next="CONSOLIDAR_SUSPENSAO_JUD">
        <batch:tasklet transaction-manager="demoiselleTransactionManager" ref="auditaTriggersTasklet" />
    </batch:step>

    <!-- #PASSO 01: CONSOLIDAR SUSPENSAO JUDICIAL -->
    <batch:step id="CONSOLIDAR_SUSPENSAO_JUD" parent="CONSOLIDAR_SUSPENSAO_JUD-PARTITIONER" next="CONSOLIDAR_CNPJ_INAPTO_CPF_CANCELADO"/>

    ...
</batch:job>
<!-- FIM JOB ===================================================================================================================== -->
```

Spring Batch



Aspectos de Desenvolvimento

✓Definição de tarefas (***jobs***), etapas (***steps***) e elementos de decisão (***decisions***) em nossas aplicações, sendo um processo batch representado por um *job*, que por sua vez consiste em um conjunto de *steps* inter-relacionados

✓Definição de um processo *batch* por meio de um fluxo completo de passos em um arquivo de configuração;

Aspectos de Execução

✓Informação de **status** para cada execução de processo;

✓Execução de processos e a **continuação de processos interrompidos** por meio de um controlador unificado;

✓Tratamento de **erros**;

✓Possibilidade de **processamento paralelo** de partes específicas do processo ou da operação como um todo;

✓Controle **transacional**.

Tipos de Passos

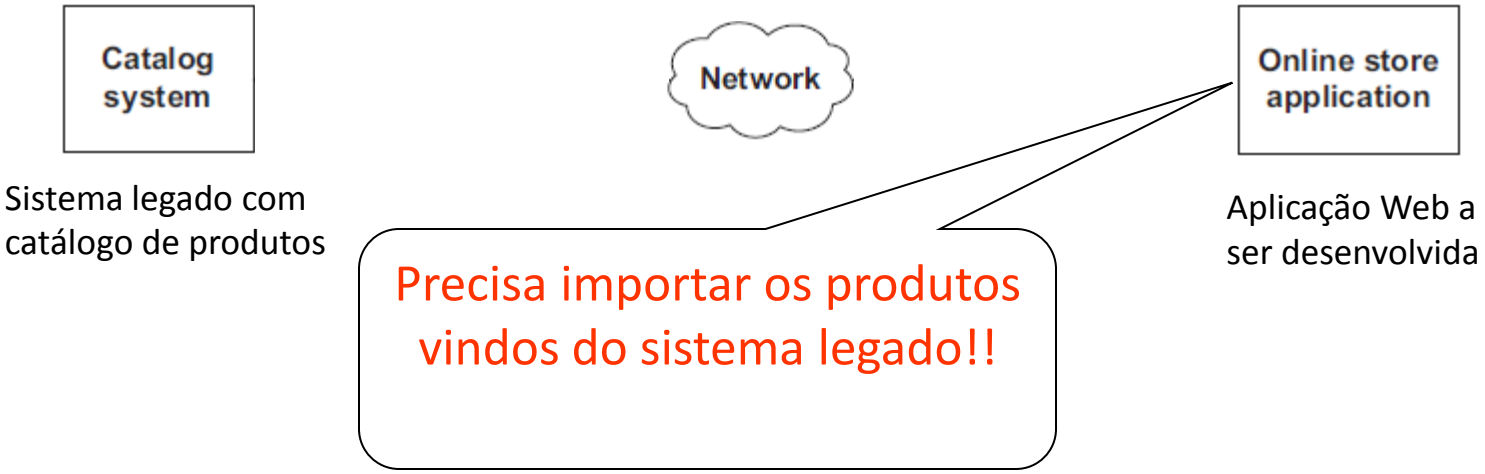
- **Os *steps* que compõem um job podem ser implementadas de duas formas diferentes**

  - **Passo Simples (*Tasklet*)**
    - Delega para apenas uma classe a execução de todo o passo

  - **Passo baseado em *Chunks***
    - Passo explicitamente separado em 3 fases:
      1. **Leitura** dos dados;
      2. **Processamento** dos dados;
      3. **Escrita** dos dados.

Exemplo Prático

- Estudo de caso

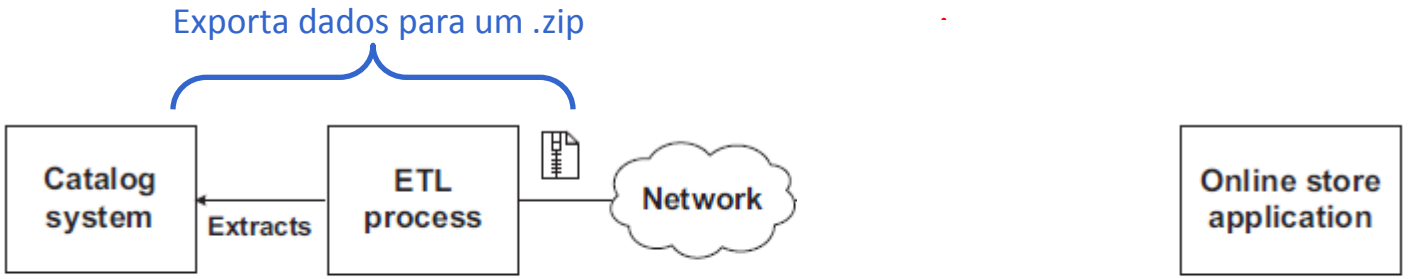


Sistema legado com catálogo de produtos

Aplicação Web a ser desenvolvida

Exemplo Prático

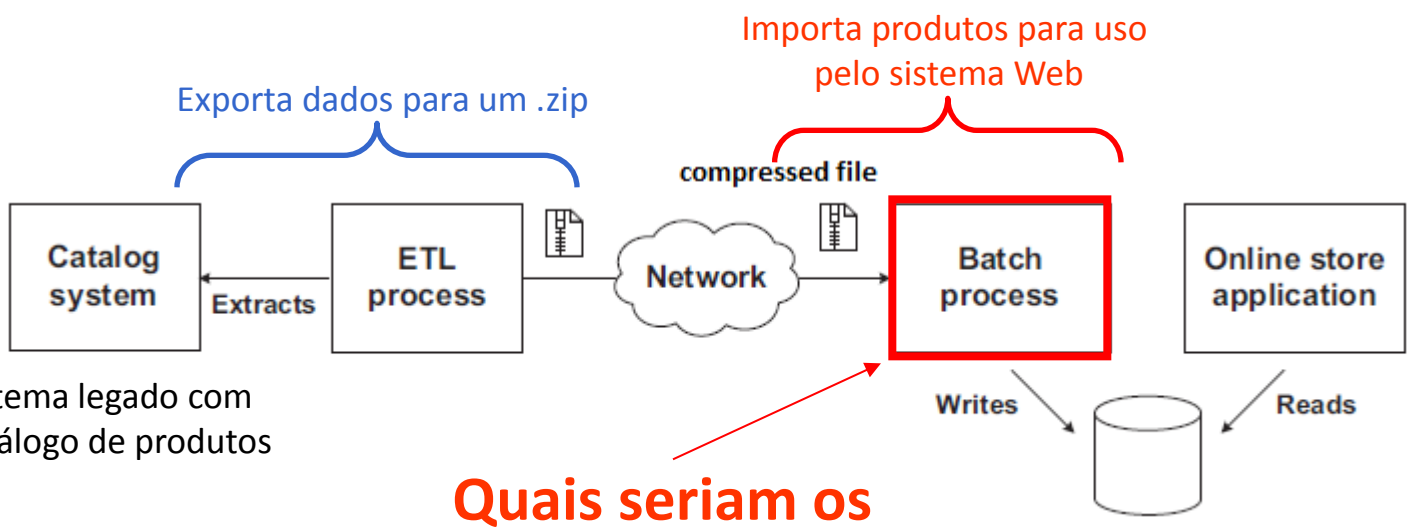
- ## Estudo de caso



Sistema legado com catálogo de produtos

Exemplo Prático

- ## Estudo de caso

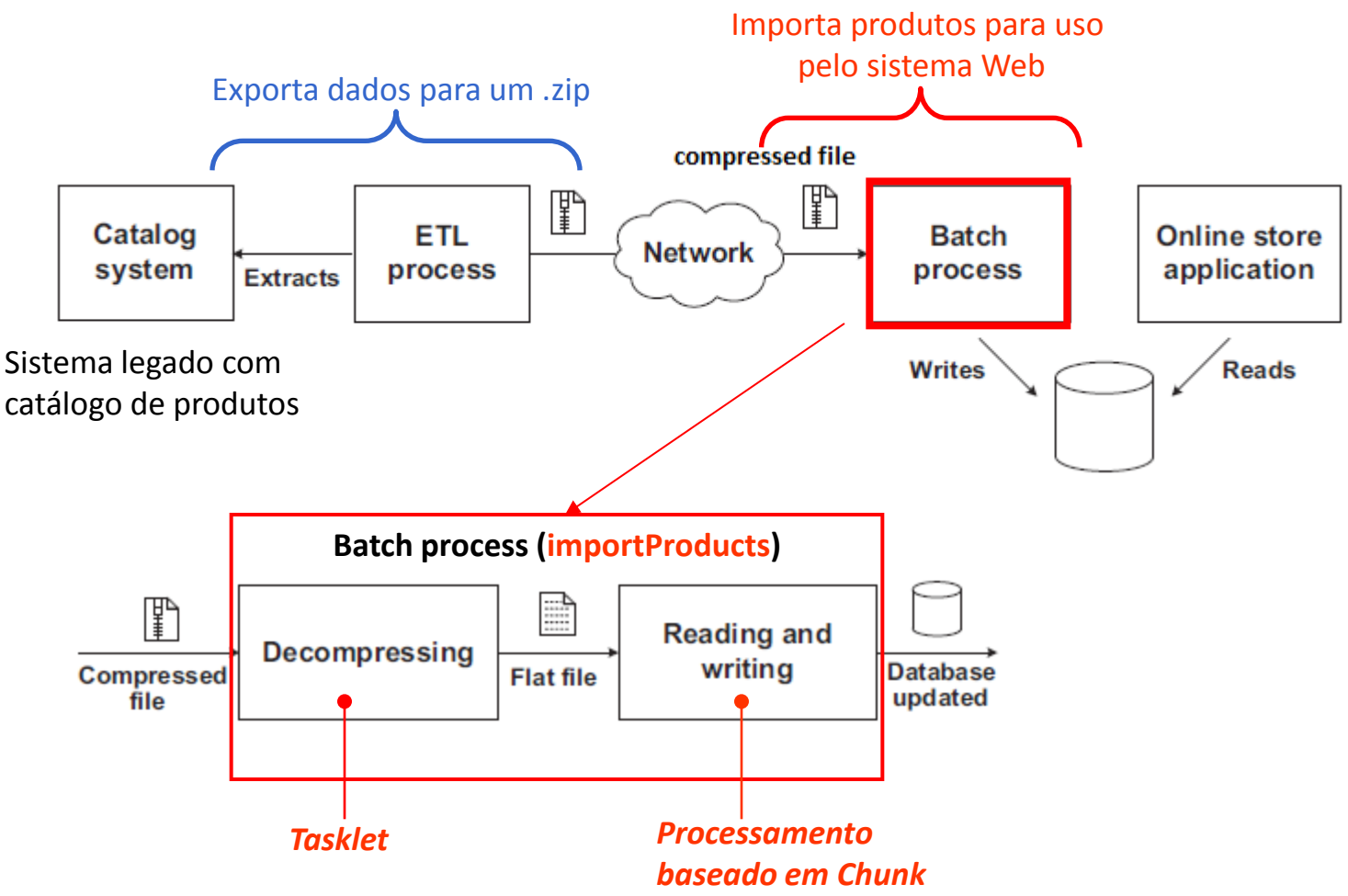


Sistema legado com catálogo de produtos

Quais seriam os passos desse job?

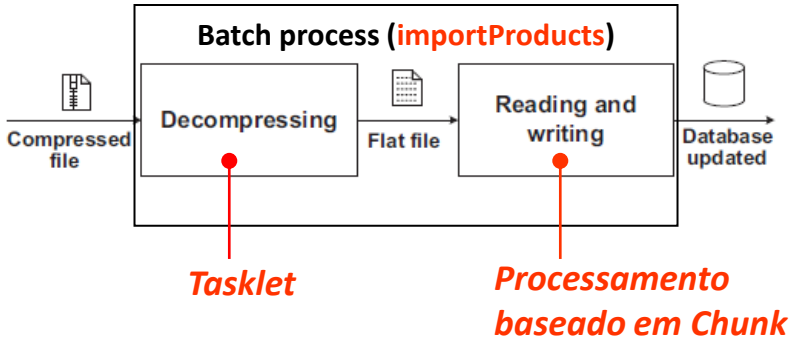
Exemplo Prático

- ## Estudo de caso



Sistema legado com catálogo de produtos

Exemplo Prático



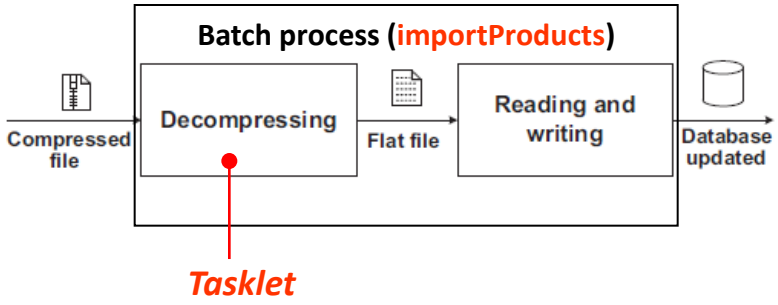
- Configuração do Job

Job importProducts

```
<job id="importProducts"
     xmlns="http://www.springframework.org/schema/batch">
  <step id="decompress" next="readWriteProducts">
    <tasklet ref="decompressTasklet" />
  </step>
  <step id="readWriteProducts">
    <tasklet>
      <chunk reader="reader" writer="writer" commit-interval="100" />
    </tasklet>
  </step>
</job>
```

Tasklet

Exemplo Prático



Configuração do Job

```
<job id="importProducts"
     xmlns="http://www.springframework.org/schema/batch">
  <step id="decompress" next="readWriteProducts">
    <tasklet ref="decompressTasklet" />
  </step>
  <step id="readWriteProducts">
    <tasklet>
      <chunk reader="reader" writer="writer" commit-interval="100" />
    </tasklet>
  </step>
</job>
```

Tasklet

```
<bean id="decompressTasklet"
      class="com.manning.sbia.ch01.batch.
      DecompressTasklet">
  <property name="inputResource"
            value="file:./input/input.zip" />
  <property name="targetDirectory"
            value="./work/output/" />
  <property name="targetFile"
            value="products.txt" />
</bean>
```

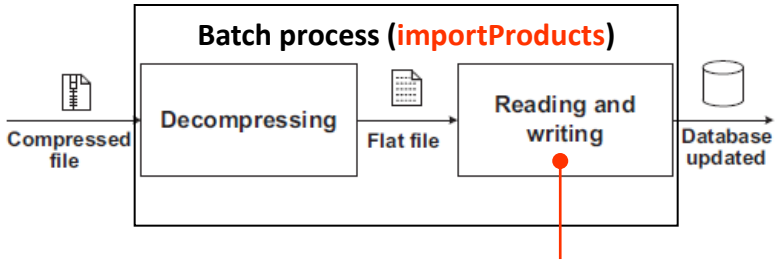
Exemplo Prático:: Implementacao

```
import org.springframework.batch.core.step.tasklet.Tasklet;
import org.springframework.batch.repeat.RepeatStatus;
import org.springframework.core.io.Resource;

public class DecompressTasklet implements Tasklet {

 private Resource inputResource;
 private String targetDirectory;
 private String targetFile;

  public RepeatStatus execute(StepContribution contribution,
       ChunkContext chunkContext) throws Exception {
    ZipInputStream zis = new ZipInputStream(
      new BufferedInputStream(
        inputResource.getInputStream()));

    File targetDirectoryAsFile = new File(
      targetDirectory);
    if(!targetDirectoryAsFile.exists()) {
      FileUtils.forceMkdir(targetDirectoryAsFile);
    }
    File target = new File(targetDirectory,targetFile);
    BufferedOutputStream dest = null;
    while(zis.getNextEntry() != null) {
      if(!target.exists()) {
        target.createNewFile();
      }
      FileOutputStream fos = new FileOutputStream(
        target
      );
      dest = new BufferedOutputStream(fos);
      IOUtils.copy(zis,dest);
      dest.flush();
      dest.close();
    }
    zis.close();
    if(!target.exists()) {
      throw new IllegalStateException(
        "Could not decompress anything from the archive!");
    }
    return RepeatStatus.FINISHED;
  }
  /* setters */
  (...)

}
```

1. Implementa a interface Tasklet
2. Declara os parâmetros da Tasklet
3. Abre arquivo
4. Cria diretório de destino (caso não exista)
5. Descompacta .zip
6. Finaliza tasklet

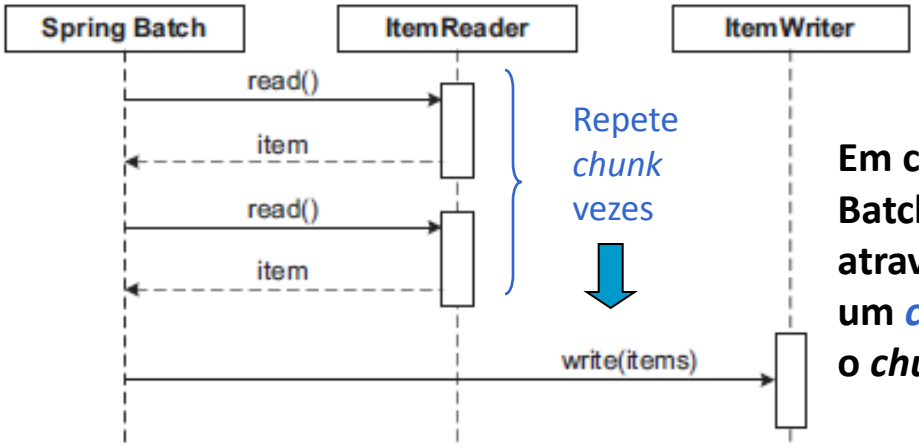
E o processamento baseado em Chunks?

Exemplo Prático



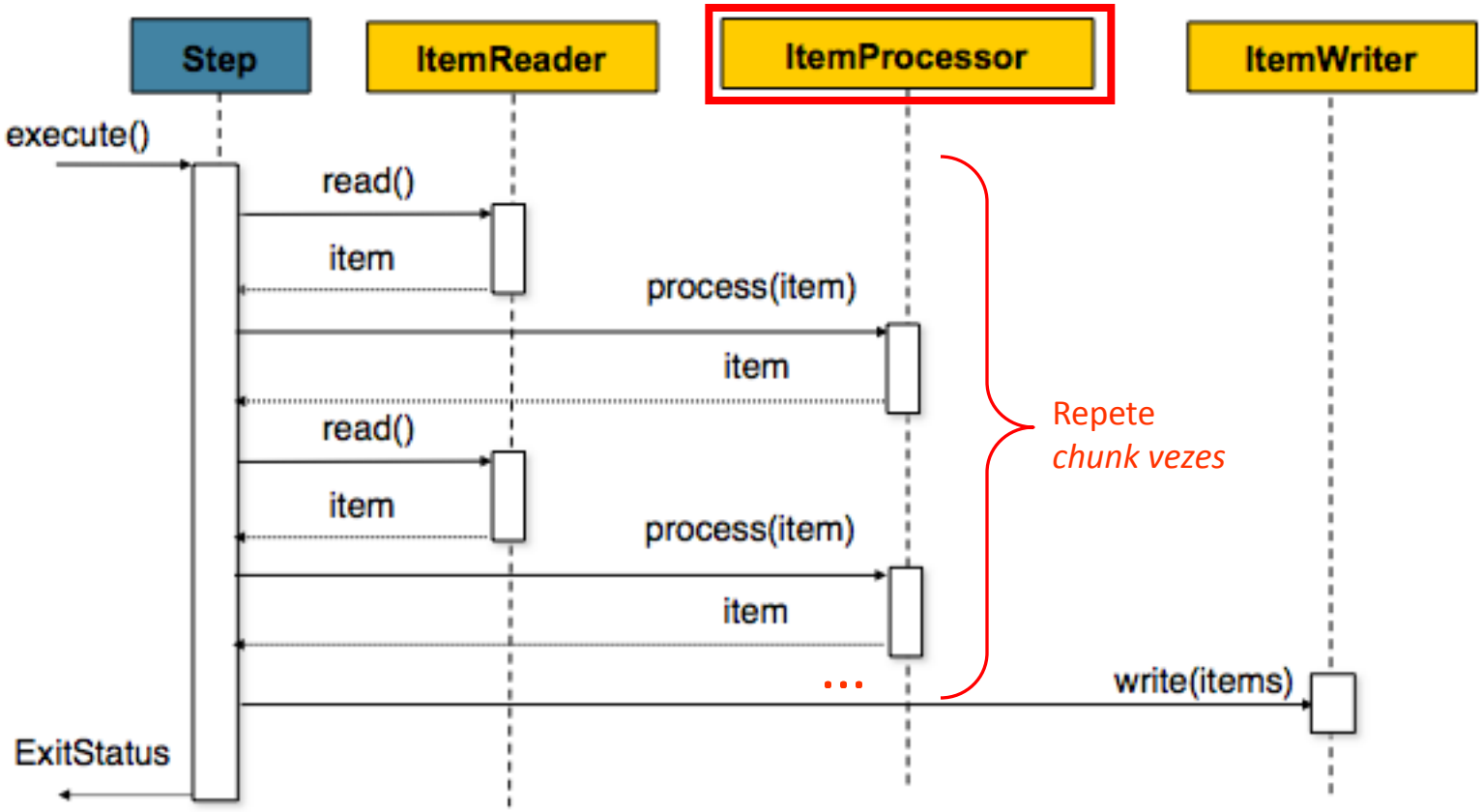
Configuração do Job

```
<job id="importProducts"
     xmlns="http://www.springframework.org/schema/batch">
  <step id="decompress" next="readWriteProducts">
    <tasklet ref="decompressTasklet" />
  </step>
  <step id="readWriteProducts">                    Read-write step
    <tasklet>
      <chunk reader="reader" writer="writer" commit-interval="100" />
    </tasklet>
  </step>
</job>
```



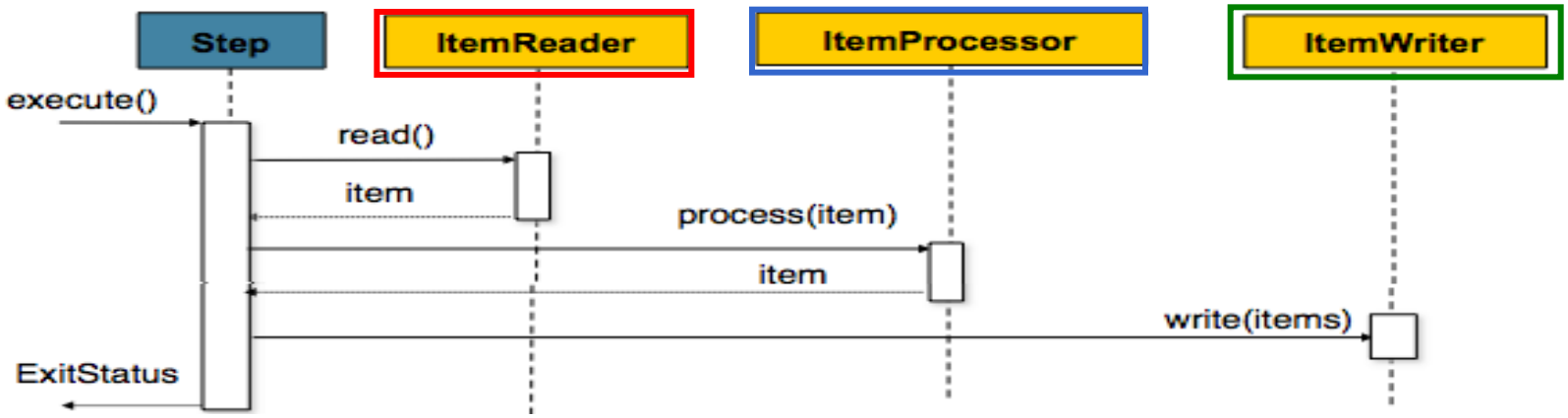
Em cenários de leitura-escrita, o Spring Batch faz a leitura de itens um a um, através do *ItemReader*, agrupa os itens em um *chunk* de uma dado tamanho, e envia o *chunk* para o *ItemWriter*.

Processamento *Chunk* + ItemProcessor



Um *ItemProcessor* pode transformar itens de entrada antes da chamada ao *ItemWriter*

Processamento *Chunk* + ItemProcessor




```
public interface ItemReader<T> {

  T read() throws Exception, UnexpectedInputException,
                  ParseException,
                  NonTransientResourceException;

}
```

```
package org.springframework.batch.item;

public interface ItemProcessor<I, O> {

  O process(I item) throws Exception;

}
```

```
package org.springframework.batch.item;

import java.util.List;

public interface ItemWriter<T> {

  void write(List<? extends T> items) throws Exception;

}
```

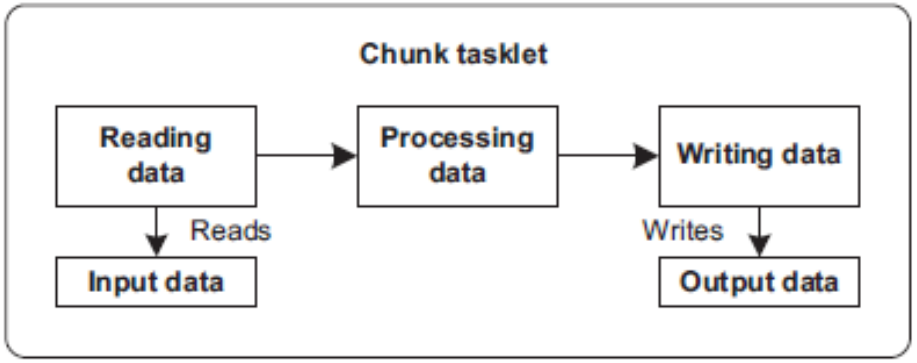
Leitura de dados...

Leitura de dados

- Interfaces *ItemReader* e *ItemStream*

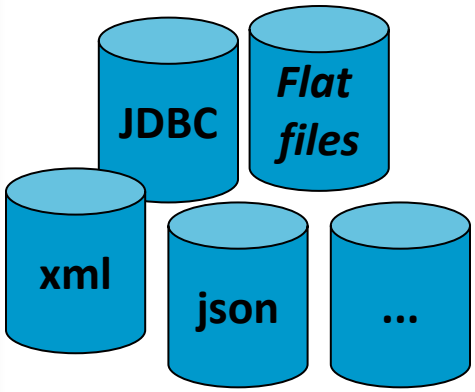
```
public interface ItemReader<T> {
  T read() throws Exception, UnexpectedInputException,
                  ParseException, NonTransientResourceException;
}
```

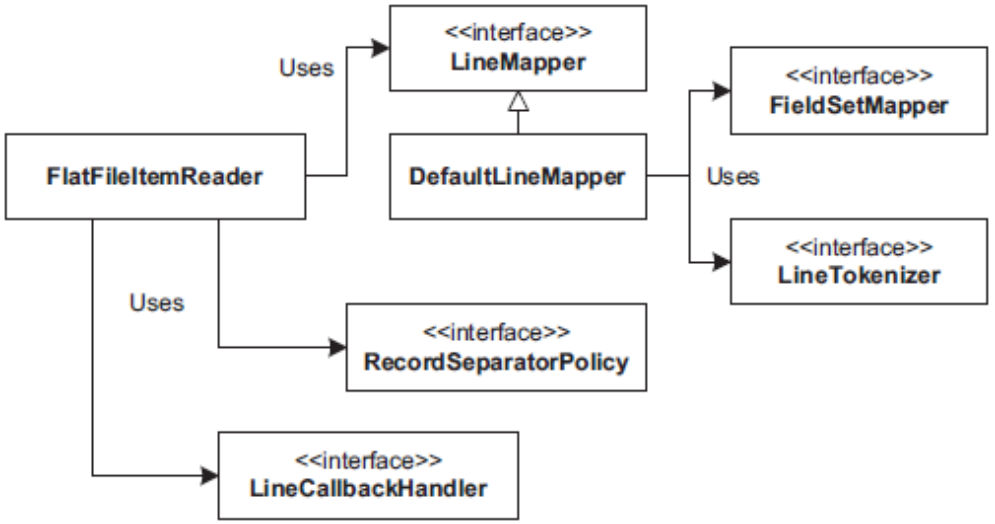
```
public interface ItemStream {
  void open(ExecutionContext executionContext)
                  throws ItemStreamException;
  void update(ExecutionContext executionContext)
                  throws ItemStreamException;
  void close() throws ItemStreamException;
}
```



Leitura de dados

- É preciso ter noção sobre como funcionam os vários tipos de leitores predefinidos





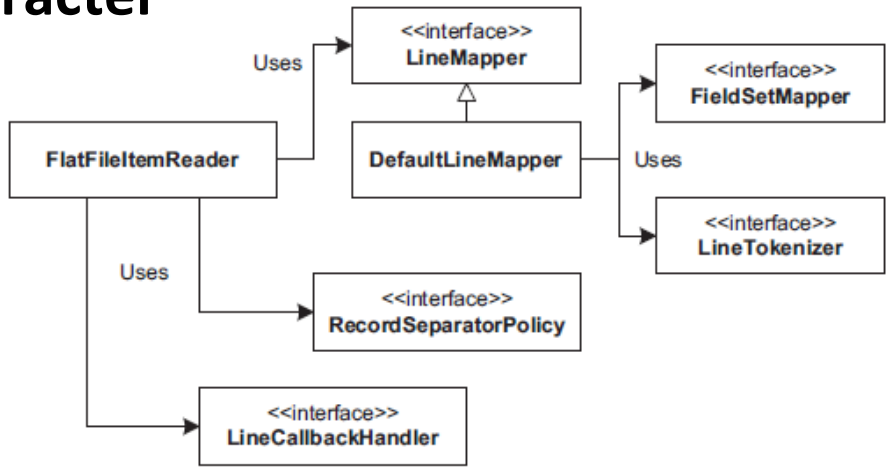
Entity	Description
`LineMapper`	Maps a data line to a data object
`FieldSetMapper`	Creates a data object from tokens; invoked by the `DefaultLineMapper` class, the default implementation of the `LineMapper` interface
`LineTokenizer`	Splits a data line into tokens; invoked by the `DefaultLineMapper` class, the default implementation of the `LineMapper` interface
`RecordSeparatorPolicy`	Identifies beginning and end of data records
`LineCallbackHandler`	Provides data lines in special cases; the common usage is for lines skipped by the `FlatFileItemReader` class during processing

Leitura de dados - exemplo

- Campos separados por caracter

```
PR....210,BlackBerry 8100 Pearl,,124.60
PR....211,Sony Ericsson W810i,,139.45
PR....212,Samsung MM-A900M Ace,,97.80
PR....213,Toshiba M285-E 14,,166.20
PR....214,Nokia 2610 Phone,,145.50
PR....215,CN Clogs Beach/Garden Clog,,190.70
PR....216,AT&T 8525 PDA,,289.20
```

datafile.txt



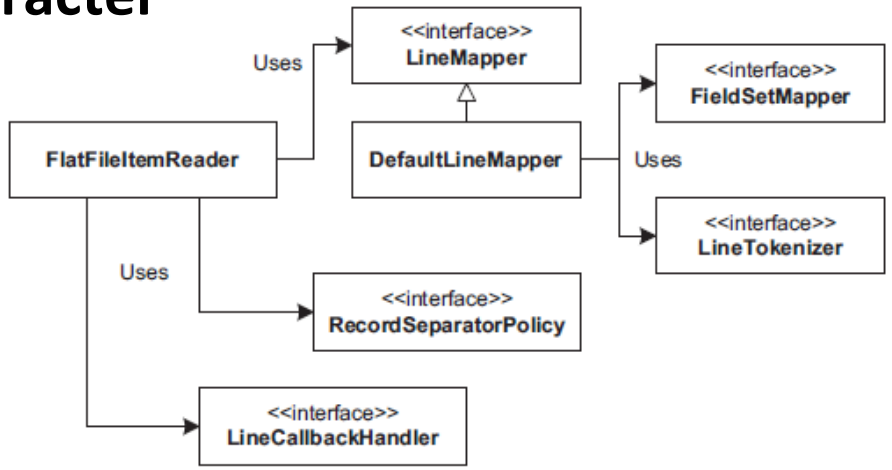

```
<bean id="productItemReader"
      class="org.springframework.batch.item.file.FlatFileItemReader">
  <property name="resource" value="datafile.txt"/>
  <property name="linesToSkip" value="1"/>
  <property name="recordSeparatorPolicy"
            ref="productRecordSeparatorPolicy"/>
  <property name="lineMapper" ref="productLineMapper"/>
</bean>

<bean id="productRecordSeparatorPolicy" class="(...)">
  (...)
</bean>

<bean id="productLineMapper" class="(...)">
  (...)
</bean>
```

Leitura de dados - exemplo

- Campos separados por caracter

```
PR....210,BlackBerry 8100 Pearl,,124.60
PR....211,Sony Ericsson W810i,,139.45
PR....212,Samsung MM-A900M Ace,,97.80
PR....213,Toshiba M285-E 14,,166.20
PR....214,Nokia 2610 Phone,,145.50
PR....215,CN Clogs Beach/Garden Clog,,190.70
PR....216,AT&T 8525 PDA,,289.20
```

datafile.txt




```
<bean id="productLineMapper"
    class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
  <property name="lineTokenizer" ref="productLineTokenizer"/>
  <property name="fieldSetMapper" ref="productFieldSetMapper"/>
</bean>

<bean id=" productLineTokenizer"
      class="org.springframework.batch.item.file
                  ➥ .transform.DelimitedLineTokenizer">
  <property name="delimiter" value=","/>
  <property name="names"
            value="id,name,description,price"/>
</bean>
```

Leitura de dados – exemplo II

- **FixedLenghtTokenizer**

```
PR....210BlackBerry 8100 Pearl                124.60
PR....211Sony Ericsson W810i                  139.45
PR....212Samsung MM-A900M Ace                  97.80
PR....213Toshiba M285-E 14                    166.20
PR....214Nokia 2610 Phone                     145.50
PR....215CN Clogs Beach/Garden Clog           190.70
PR....216AT&T 8525 PDA                        289.20
```

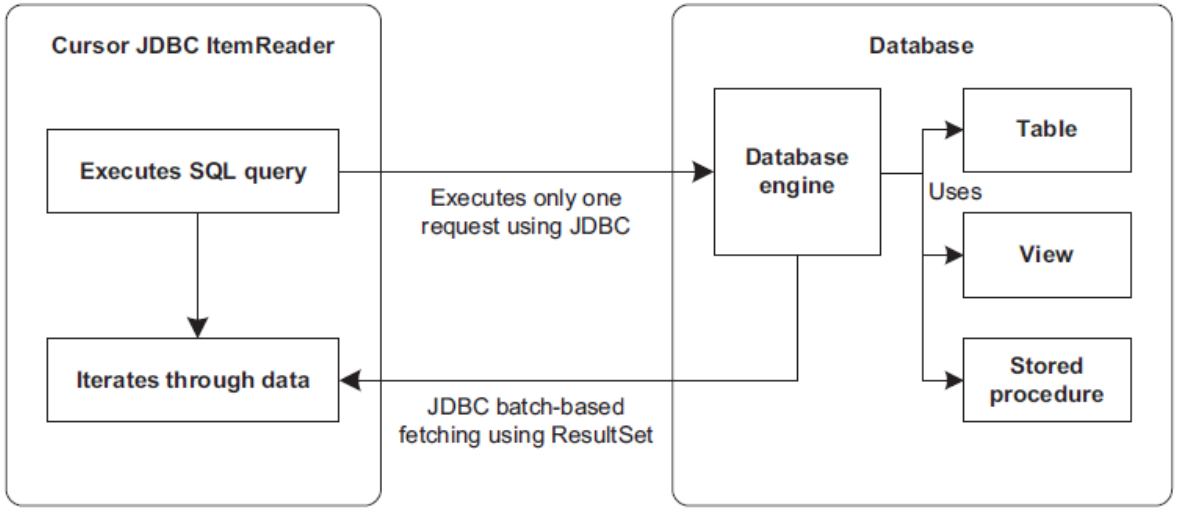
datafile.txt

Field name	Length in characters
id	9
name	26
description	15
price	6

```
<bean id=" productLineTokenizer"
      class="org.springframework.batch.item.file
                  ➥ .transform.FixedLengthTokenizer">
  <property name="columns" value="1-9,10-35,36-50,51-56"/>
  <property name="names"
            value="id,name,description,price"/>
</bean>
```

Leitura de dados – exemplo III

- JdbcCursorItemReader

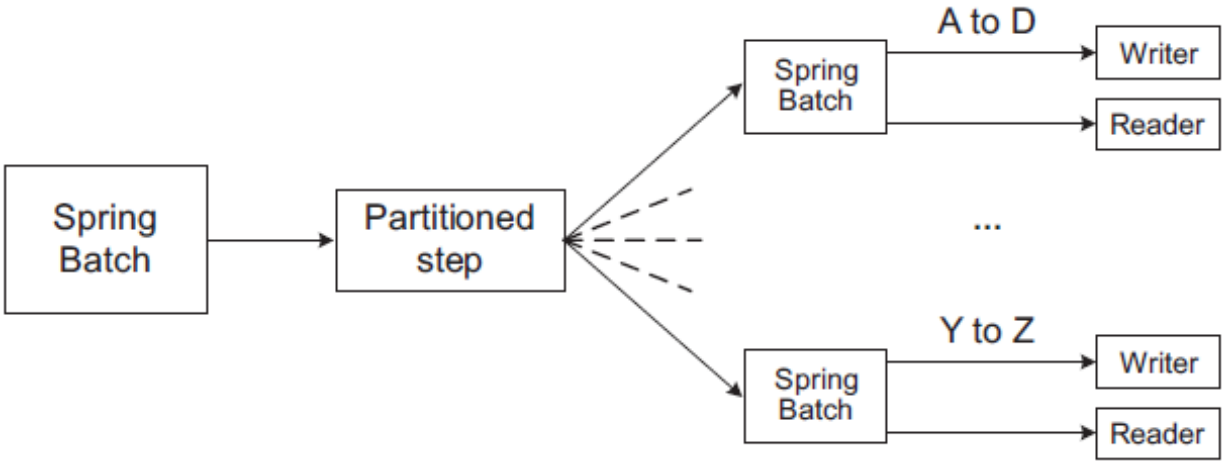



```
<bean id="productItemReader
      class="org.springfram
  <property name="dataSourc
  <property name="sql"
            value="select i
  <property name="rowMapper
            ref="productRow
</bean>

<bean id="productRowMapper
      class="com.manning.sbia.reading.jdbc.ProductRowMapper"/>
```

```
public class ProductRowMapper implements RowMapper<Product> {
  public Product mapRow(ResultSet rs, int rowNum)
      throws SQLException {
    Product product = new Product();
    product.setId(rs.getString("id"));
    product.setName(rs.getString("name"));
    product.setDescription(rs.getString("description"));
    product.setPrice(rs.getFloat("price"));
    return product;
  }
}
```

Leitura de dados

- **Particionamento**

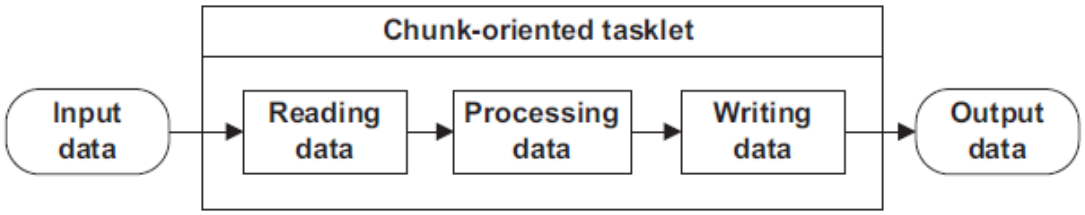


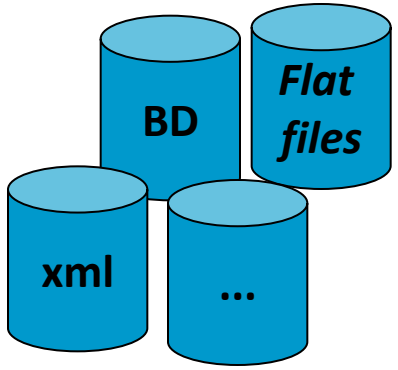
Escrita de dados...

Escrita de dados

- Também é preciso ter noção sobre como funcionam os *writers* predefinidos

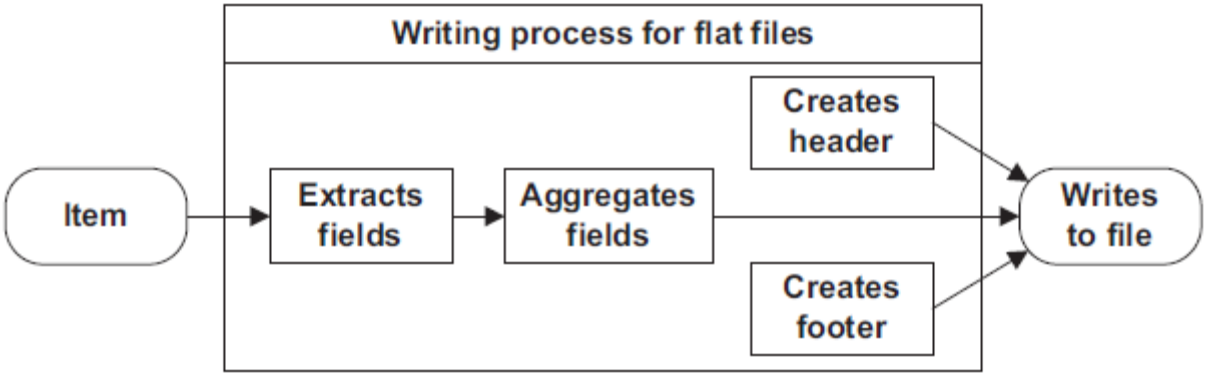
```
<tasklet>
  <chunk reader="itemReader" writer="itemWriter" commit-interval="100"/>
</tasklet>
```





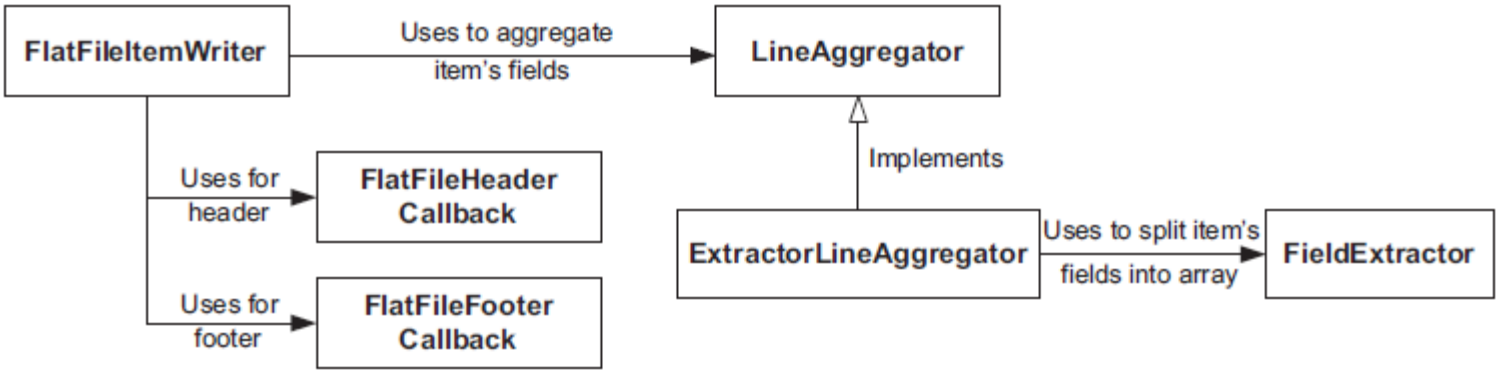
Escrita de dados - Exemplo

- Escrevendo em arquivo
  - Escrever cabeçalho (opcional)
  - Extrair campos de cada item e agregá-los para produzir uma linha
  - Escrever rodapé (opcional)



Escrita de dados - Exemplo

- Escrevendo em arquivo

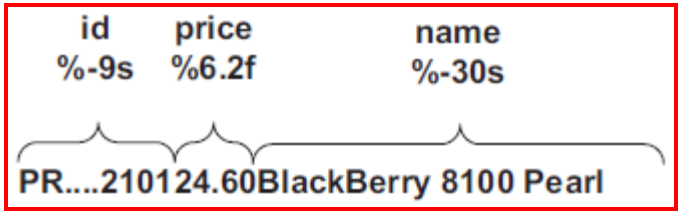


Type	Description
`LineAggregator`	Creates a `String` representation of an object
`ExtractorLineAggregator`	Implements `LineAggregator` to extract field data from a domain object
`FieldExtractor`	Creates an array containing item parts
`FlatFileHeaderCallback`	Callback interface to write a header
`FlatFileFooterCallback`	Callback interface to write a footer

Escrita de dados - Exemplo

- *FormatterLineAggregator*

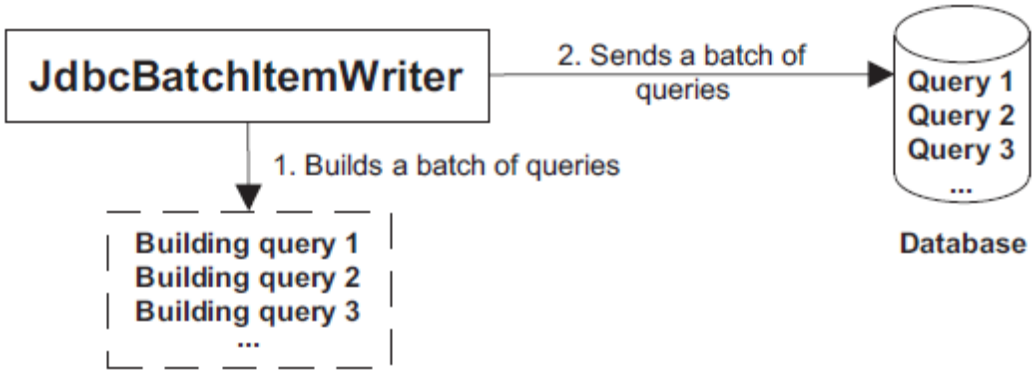
```
PR....210124.60BlackBerry 8100 Pearl
PR....211139.45Sony Ericsson W810i
PR....212 97.80Samsung MM-A900M Ace
PR....213166.20Toshiba M285-E 14
```




```
<bean id="productItemWriter"
      class="org.springframework.batch.item.file.FlatFileItemWriter">
  <property name="resource"
    value="file:target/outputs/fixedwidth-beanwrapperextractor.txt"/>
  <property name="lineAggregator">
    <bean class="org.springframework.batch.item.file.transform.
  ➥ FormatterLineAggregator">
      <property name="fieldExtractor">
        <bean class="org.springframework.batch.item.file.transform.
  ➥ BeanWrapperFieldExtractor">
          <property name="names" value="id,price,name" />
        </bean>
      </property>
      <property name="format" value="%-9s%6.2f%-30s" />
    </bean>
  </property>
</bean>
```

Escrita de dados - Exemplo

- *JdbcBatchItemWriter*
  - Enviar um **lote de comandos SQL é MUITO mais eficiente** do que enviar um sql por vez



Escrita de dados - Exemplo

- *JdbcBatchItemWriter*
  - Enviar um **batch de comandos SQL é MUITO mais eficiente** do que enviar um sql por vez

```
<bean id="productItemWriter"
      class="org.springframework.batch.item.database.
  ➥ JdbcBatchItemWriter">
  <property name="assertUpdates" value="true" />
  <property name="itemSqlParameterSourceProvider">
    <bean class="org.springframework.batch.item.database.
  ➥ BeanPropertyItemSqlParameterSourceProvider" />
  </property>
  <property name="sql"
    value="INSERT INTO PRODUCT (ID, NAME, PRICE)
           VALUES(:id, :name, :price)" />
  <property name="dataSource" ref="dataSource" />
</bean>
```

Property	Type	Description
assertUpdates	boolean	Whether to throw an exception if at least one item doesn't update or delete a row; defaults to true
itemPreparedStatementSetter	ItemPreparedStatementSetter<T>	SQL statement parameter values from ? positional parameter markers
itemSqlParameterSourceProvider	ItemSqlParameterSourceProvider<T>	SQL statement parameter values from named parameters
sql	String	SQL statement to execute

Escrita de dados - Exemplo

- *ItemPreparedStatementSetter*

```
<bean id="productItemWriter" class="org.springframework.batch.item.
  ➥ database.JdbcBatchItemWriter">
  <property name="assertUpdates" value="true" />
  <property name="itemPreparedStatementSetter">
    <bean class="com.manning.sbia.ch06.database.
  ➥ ProductItemPreparedStatementSetter" />
  </property>
  <property name="sql"
    value="INSERT INTO PRODUCT (ID, NAME, PRICE)
  ➥ VALUES(?, ?, ?)" />
  <property name="dataSource" ref="dataSource" />
</bean>
```

```
public class ProductItemPreparedStatementSetter
        implements ItemPreparedStatementSetter<Product> {
  @Override
  public void setValues(Product item,
                        PreparedStatement ps) throws SQLException {
    ps.setString(1, item.getId());
    ps.setString(2, item.getName());
    ps.setBigDecimal(3, item.getPrice());
  }
}
```

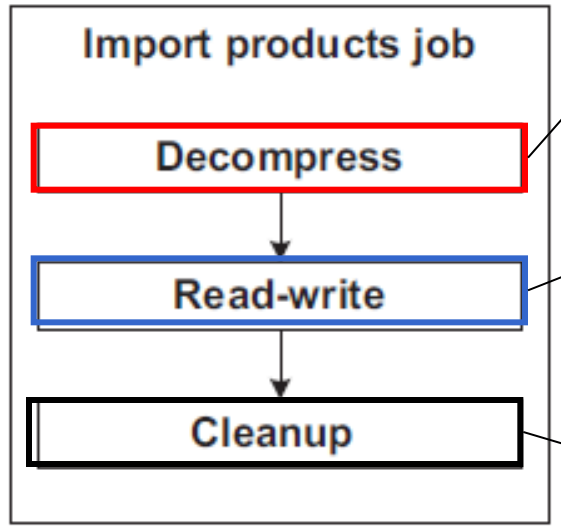
Seta os parâmetros SQL

E o hibernate?

Decidindo o fluxo de execução...

Fluxo de execução

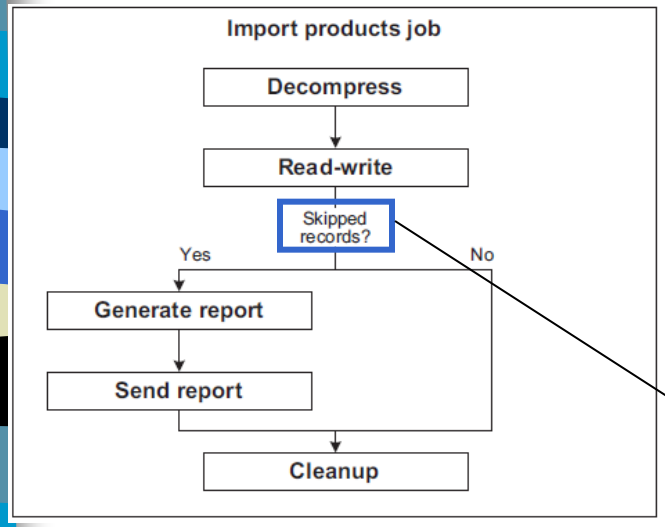
- Linear (propriedade **next**)




```
<job id="importProductsJob">
  <step id="decompress" next="readWrite">
    <tasklet ref="decompressTasklet" />
  </step>
  <step id="readWrite" next="clean">
    <tasklet>
      <chunk reader="reader" writer="writer"
        commit-interval="100" />
    </tasklet>
  </step>
  <step id="clean">
    <tasklet ref="cleanTasklet" />
  </step>
</job>
```

Fluxo de execução

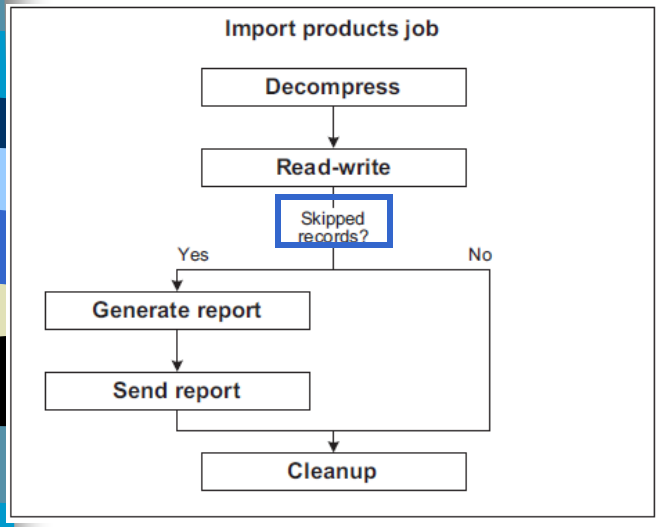
- Fluxo não linear (elemento *<decision>*)




```
<job id="importProductsJob"
     xmlns="http://www.springframework.org/schema/batch">
  <step id="decompress" next="readWrite">
    <tasklet ref="decompressTasklet" />
  </step>
  <step id="readWrite" next="skippedDecision">
   <tasklet>
     <chunk reader="reader" writer="writer" commit-interval="100" />
   </tasklet>
  </step>
  <decision id="skippedDecision"
            decider="skippedDecider">
    <next on="SKIPPED" to="generateReport"/>
    <next on="*" to="clean" />
  </decision>
  <step id="generateReport" next="sendReport">
    <tasklet ref="generateReportTasklet" />
  </step>
  <step id="sendReport" next="clean">
    <tasklet ref="sendReportTasklet" />
  </step>
  <step id="clean">
    <tasklet ref="cleanTasklet" />
  </step>
</job>

<bean id="skippedDecider"
      class="com.manning.sbia.ch02.structure.
  ➥ SkippedDecider" />
```

Lógica de decisão

- Fluxo não linear (elemento *<decision>*)

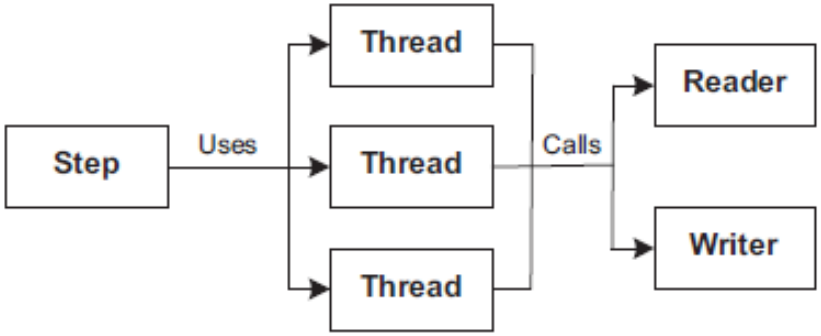



```
  <decision id="skippedDecision"
            decider="skippedDecider">
    <next on="SKIPPED" to="generateReport"/>
    <next on="*" to="clean" />
  </decision>
  <step id="generateReport" next="sendReport">
    <tasklet ref="generateReportTasklet" />
  </step>
  <step id="sendReport" next="clean">
    <tasklet ref="sendReportTasklet" />
  </step>
  <step id="clean">
    <tasklet ref="cleanTasklet" />
  </step>
</job>

<bean id="skippedDecider"
      class="com.manning.sbia.ch02.structure.
      ➥ SkippedDecider" />
```

```
public class SkippedDecider implements JobExecutionDecider {
        @Override
        public FlowExecutionStatus decide(JobExecution jobExecution,
                        StepExecution stepExecution) {
                return stepExecution.getSkipCount() == 0 ? FlowExecutionStatus.COMPLETED : new FlowExecutionStatus("SKIPPED");
        }

}
```

Múltiplas *threads*

- Definindo um ***taskExecutor***

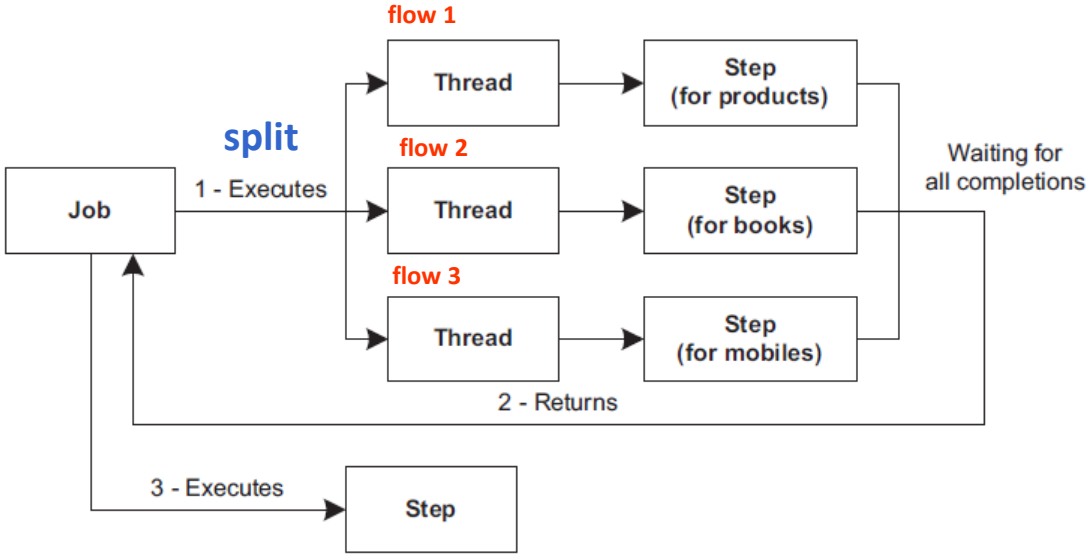



```
<batch:job id="importProductsMultiThreadedJob">
  <batch:step id="readWriteProductsMultiThreadedStep">
    <batch:tasklet task-executor="taskExecutor">
      <batch:chunk reader="reader" writer="writer" commit-interval="10"/>
    </batch:tasklet>
  </batch:step>
</batch:job>
```

```
<bean id="taskExecutor"
  class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
  <property name="corePoolSize" value="5"/>
  <property name="maxPoolSize" value="5"/>
</bean>
```

Paralelização

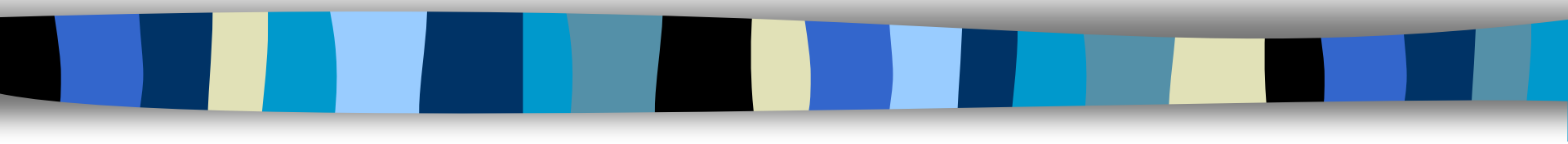
- Elementos ***split*** ***e flow***



flow

```
<batch:job id="importProductsJob">
  <batch:step id="decompress" next="readWrite">
    <batch:tasklet ref="decompressTasklet"/>
  </batch:step>
  <batch:split id="readWrite" next="moveProcessedFiles">
    <batch:flow>
      <batch:step id="readWriteBookProduct"/>
    </batch:flow>
    <batch:flow>
      <batch:step id="readWriteMobileProduct"/>
    </batch:flow>
  </batch:split>
  <batch:step id="moveProcessedFiles">
    <batch:tasklet ref="moveProcessedFilesTasklet" />
  </batch:step>
</batch:job>
```

Transações no Spring Batch

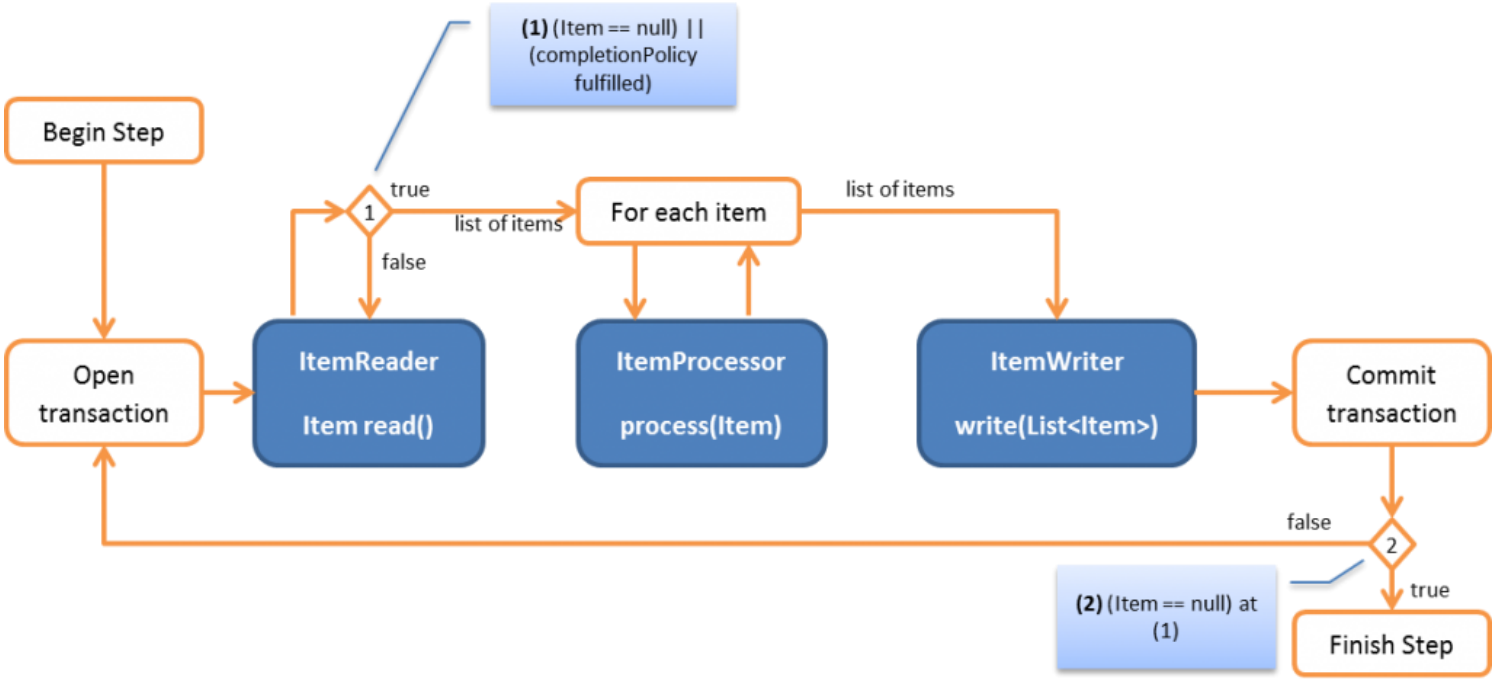


Conceitos básicos

- Gerenciamento de transações != aplicações online
  - Devemos evitar uma transação para todo o job
  - commits parciais devem ser feitos (onde?)
- E como lidar com falhas?
  - Saberemos que dados não foram alterados?
  - É possível reiniciar o job de onde parou?
  - E se for preciso pular os items que deram falha e continuar o processamento?

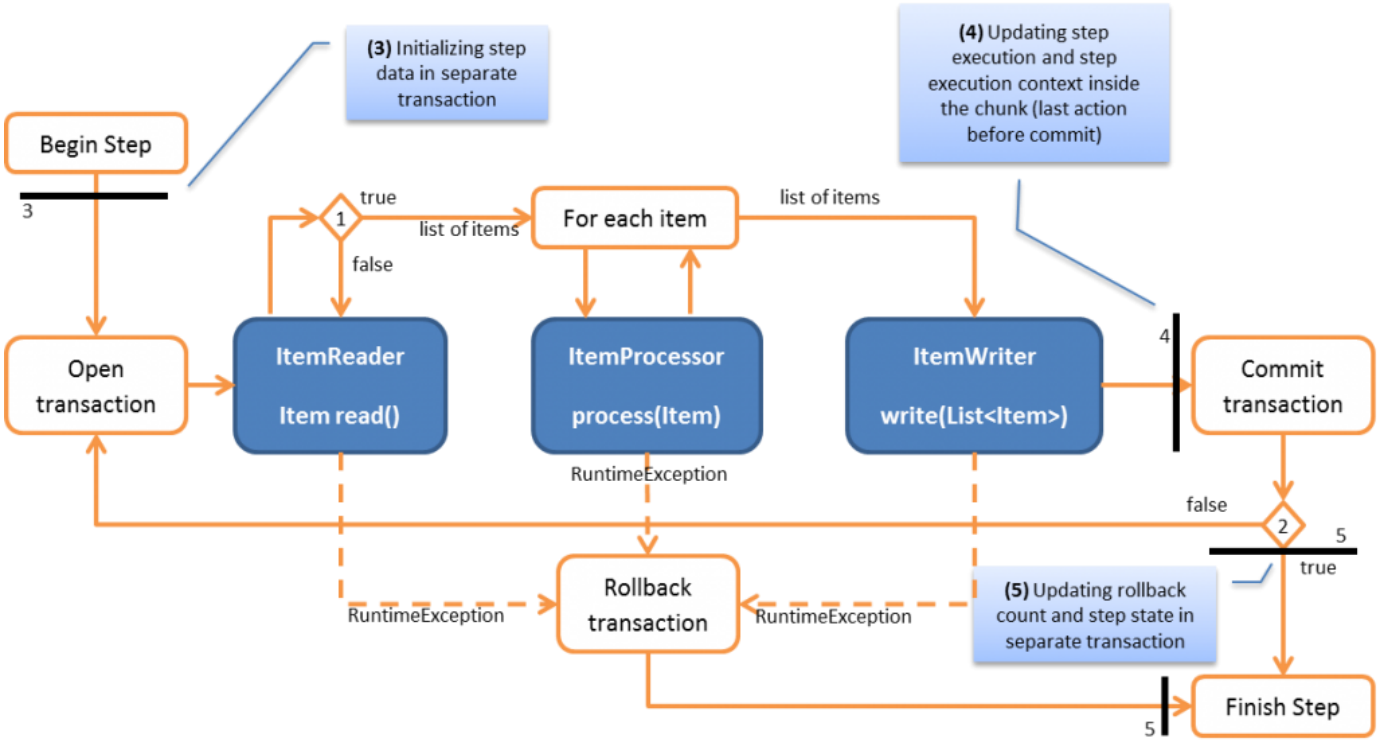
Dica: Esteja familizarizado com os conceitos de **transações do Spring framework**!!

Processamento orientado a *chunks*



- Cada *chunk* roda em sua própria transação
- O tamanho do chunk é determinado por uma *CompletionPolicy* (*SimpleCompletionPolicy* é o default)
- Caso queria customizar use o atributo chunk-completion-policy
- Informações de contexto e de execução são salvas no jobRepository

E se uma falha acontecer?



- caso nenhuma política de skip ou retry tiver sido definida
  - **Rollback** na transação
  - Passo (step) marcado como **FAILED**
  - Todo o job marcado como **FAIL**

Nem todas as exceções causam *rollback*

- Elemento *no-rollback-exceptions*

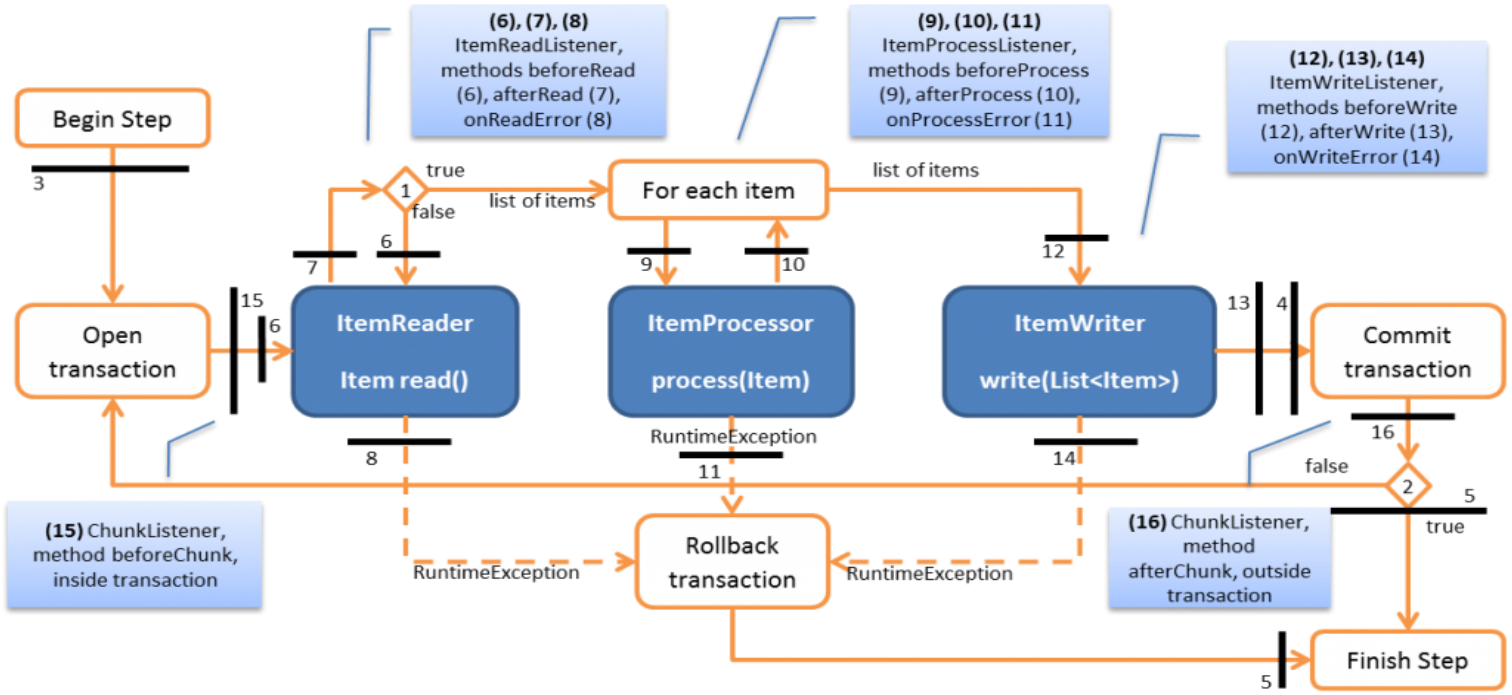
```
<batch:tasklet>
  <batch:chunk ... />
  <batch:no-rollback-exception-classes>
    <batch:include class="de.codecentric.MyRuntimeException"/>
  </batch:no-rollback-exception-classes>
</batch:tasklet>
```

Atributos de uma transação

- *propagation type, **isolation level*** e *timeout*
  - *isolation level:* determina como e quando as operações feitas por uma transação se tornam visíveis a outras
- Exemplo:

```
<job id="importProductsJob">
  <step id="importProductsStep">
    <tasklet>
      <chunk reader="reader" writer="writer" commit-interval="100" />
      <transaction-attributes
        isolation="READ_UNCOMMITTED" />
    </tasklet>
  </step>
</job>
```

Listeners e Restart



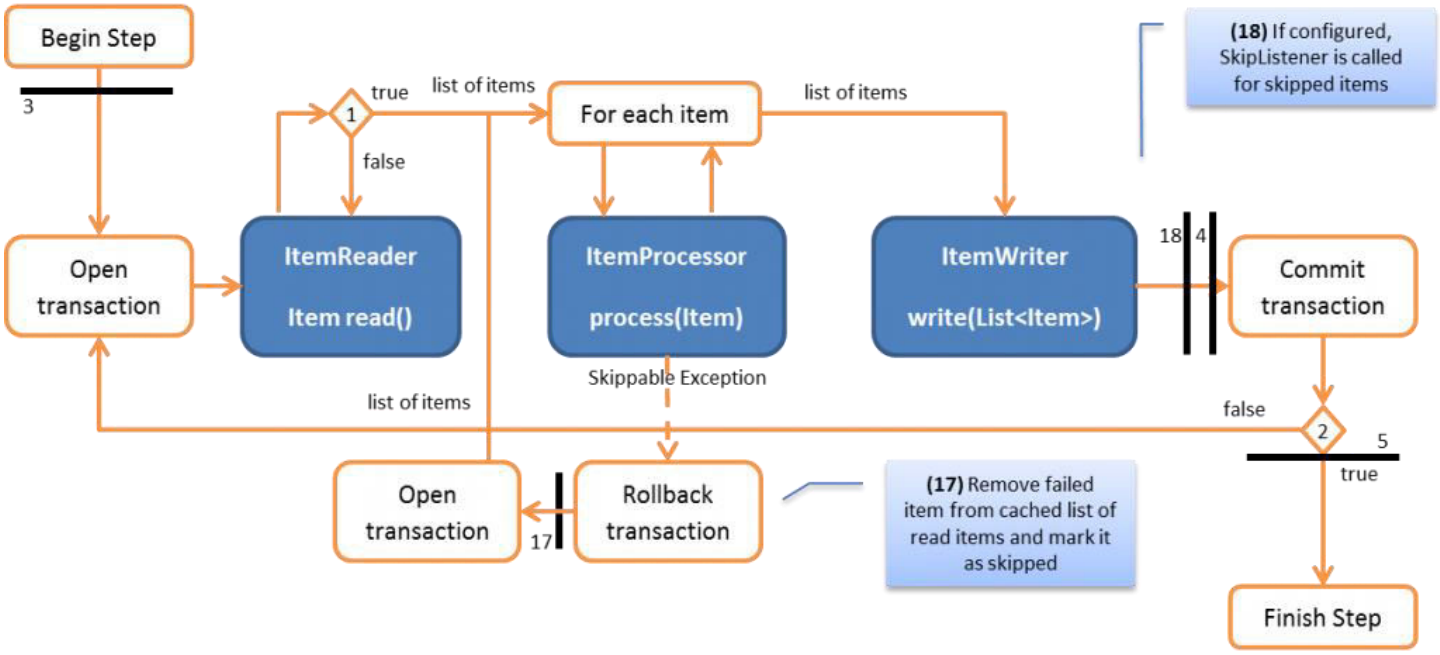
- Vários *listeners* disponíveis
  - *ItemReaders*, *ItemProcessors*, *ItemWriters*, *JobExecutionListener*, *StepExecutionListener*, *ChunkListener*, *SkipListener*
- *AbstractItemCountingItemStreamItemReader*: armazena contador no contexto de execução, que informa qual o item falhou

Skip

- **Skip**
  - Especifica tipos de exceções e número máximo de itens pulados
  - O job não falha quando uma *"skippable exception"* ocorre e continua o processamento

```
<batch:tasklet> <batch:chunk reader="myItemReader"
writer="myItemWriter" commit-interval="20" skip-limit="15">
 <batch:skippable-exception-classes>
   <batch:include class="de.codecentric.MySkippableException" />
</batch:skippable-exception-classes>
</batch:chunk>
</batch:tasklet>
```

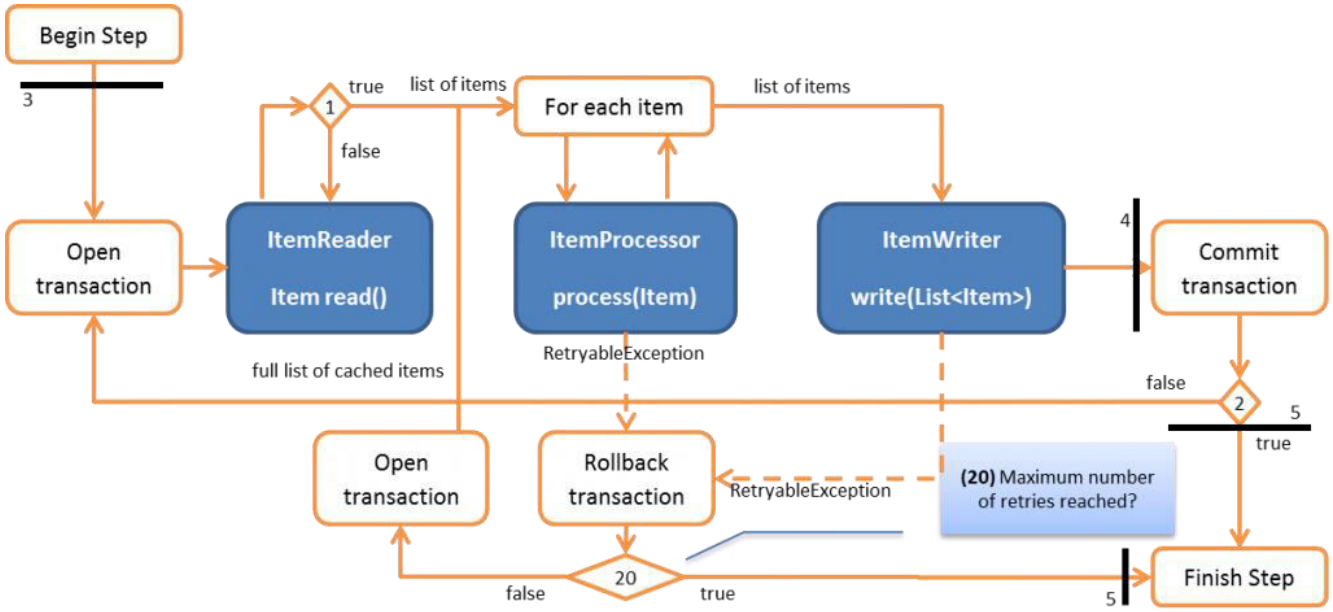
Skip



- Políticas de skip (*SkipPolicy*) podem ser especificadas

Retry

- *RetryPolicy* também pode ser especificada




```
<batch:tasklet>
<batch:chunk reader="myItemReader" writer="myItemWriter" commit-
interval="50" retry-limit="20">
    <batch:retryable-exception-classes>
        <batch:include class="de.codecentric.MyRetryableException" />
    </batch:retryable-exception-classes>
</batch:chunk>
</batch:tasklet>
```

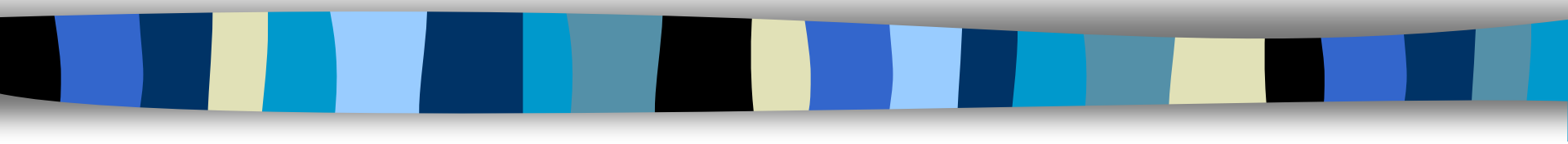
Dúvidas?

Dúvidas?

Dúvidas?

Dúvidas?

Desmistificando o *"Spring Batch"*



Vanilson Burégio