

Spring Batch Workshop (advanced)

Arnaud Cogoluègnes

Consultant with Zenika, Co-author Spring Batch in Action

August 9, 2011

Outline

Overview

XML file reading

Item enrichment

File reading partitioning

File dropping launching

Database reading partitioning

Complex flow

Atomic processing

Overview

- ▶ This workshop highlights advanced Spring Batch features
- ▶ Problem/solution approach
 - ▶ A few slides to cover the feature
 - ▶ A project to start from, just follow the TODOs
- ▶ Prerequisites
 - ▶ Basics about Java and Java EE
 - ▶ Spring: dependency injection, enterprise support
 - ▶ Spring Batch: what the first workshop covers
- ▶ <https://github.com/acogoluegnes/Spring-Batch-Workshop>

License

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/> or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

- ▶ Problem: reading items from a XML file and sending them to another source (e.g. database)
- ▶ Solution: using the `StaxEventItemReader`

Spring Batch's support for XML file reading

- ▶ Spring Batch has built-in support for XML files
 - ▶ Through the `StaxEventItemReader` for reading
- ▶ The `StaxEventItemReader` handles I/O for efficient XML processing
- ▶ 2 main steps:
 - ▶ Configuring the `StaxEventItemReader`
 - ▶ Configuring a Spring OXM's `Unmarshaller`

The usual suspects

```
<?xml version="1.0" encoding="UTF-8"?>
<contacts>
  <contact>
    <firstname>De-Anna</firstname>
    <lastname>Raghunath</lastname>
    <birth>2010-03-04</birth>
  </contact>
  <contact>
    <firstname>Susy</firstname>
    <lastname>Hauerstock</lastname>
    <birth>2010-03-04</birth>
  </contact>
  (...)
</contacts>
```



```
public class Contact {

    private Long id;
    private String firstname, lastname;
    private Date birth;

    (...)
}
```

The StaxEventItemReader configuration

```
<bean id="reader" class="org.springframework.batch.item.xml.StaxEventItemReader">
  <property name="fragmentRootElementName" value="contact" />
  <property name="unmarshaller">
    <bean class="org.springframework.oxm.xstream.XStreamMarshaller">
      <property name="aliases">
        <map>
          <entry key="contact" value="com.zenika.workshop.springbatch.Contact" />
        </map>
      </property>
      <property name="converters">
        <bean class="com.thoughtworks.xstream.converters.basic.DateConverter">
          <constructor-arg value="yyyy-MM-dd" />
          <constructor-arg value="array" />
          <constructor-arg value="true" />
        </bean>
      </property>
    </bean>
  </property>
  <property name="resource" value="classpath:contacts.xml" />
</bean>
```

- ▶ NB: Spring OXM supports XStream, JAXB2, etc.

Going further...

- ▶ `StaxEventItemWriter` to write XML files
- ▶ Spring OXM's support for other marshallers
- ▶ Skipping badly formatted lines

- ▶ Problem: I want to enrich read items with a Web Service before they get written
- ▶ Solution: implement an `ItemProcessor` to make the Web Service call

Use case

- ▶ Reading contacts from a flat file
- ▶ Enriching the contact with their social security number
- ▶ Writing the whole contact in the database

The input file and the domain object

```
1,De-Anna ,Raghunath,2010-03-04  
2,Susy ,Hauerstock,2010-03-04  
3,Kiam ,Whitehurst,2010-03-04  
4,Alecia ,Van Holst,2010-03-04  
5,Hing ,Senecal,2010-03-04
```

► NB: no SSN!

```
public class Contact {  
  
    private Long id;  
    private String firstname, lastname;  
    private Date birth;  
    private String ssn;  
    (...)  
}
```

The Web Service

- ▶ It can be any kind of Web Service (SOAP, REST)
- ▶ Our Web Service
 - ▶ URL:
`http://host/service?firstname=John&lastname=Doe`
 - ▶ It returns

```
<contact>
  <firstname>John</firstname>
  <lastname>Doe</lastname>
  <ssn>987-65-4329</ssn>
</contact>
```

The ItemProcessor implementation

```
package com.zenika.workshop.springbatch;

import javax.xml.transform.dom.DOMSource;
import org.springframework.batch.item.ItemProcessor;
import org.springframework.web.client.RestTemplate;
import org.w3c.dom.NodeList;

public class SsnWebServiceItemProcessor implements
    ItemProcessor<Contact, Contact> {

    private RestTemplate restTemplate = new RestTemplate();
    private String url;

    @Override
    public Contact process(Contact item) throws Exception {
        DOMSource source = restTemplate.getForObject(url, DOMSource.class,
            item.getFirstname(), item.getLastname());
        String ssn = extractSsnFromXml(item, source);
        item.setSsn(ssn);
        return item;
    }

    private String extractSsnFromXml(Contact item, DOMSource source) {
        // some DOM code
    }
    (...)
}
```

Configuring the SsnWebServiceItemProcessor

```
<batch:job id="itemEnrichmentJob">
  <batch:step id="itemEnrichmentStep">
    <batch:tasklet>
      <batch:chunk reader="reader" processor="processor" writer="writer"
        commit-interval="3"/>
    </batch:tasklet>
  </batch:step>
</batch:job>

<bean id="processor"
  class="com.zenika.workshop.springbatch.SsnWebServiceItemProcessor">
  <property name="url"
    value="http://localhost:8085/?firstname={firstname}&lastname={lastname}" />
</bean>
```

But my Web Service has a lot of latency!

- ▶ The Web Service call can benefit from multi-threading
- ▶ Why not spawning several processing at the same time?
- ▶ We could wait for the completion in the `ItemWriter`
- ▶ Let's use some asynchronous `ItemProcessor` and `ItemWriter`
 - ▶ Provided in the Spring Batch Integration project

Using async ItemProcessor and ItemWriter

- ▶ This is only about wrapping

```
<bean id="processor"
      class="org.springframework.batch.integration.async.AsyncItemProcessor">
  <property name="delegate" ref="processor" />
  <property name="taskExecutor" ref="taskExecutor" />
</bean>

<bean id="writer"
      class="org.springframework.batch.integration.async.AsyncItemWriter">
  <property name="delegate" ref="writer" />
</bean>

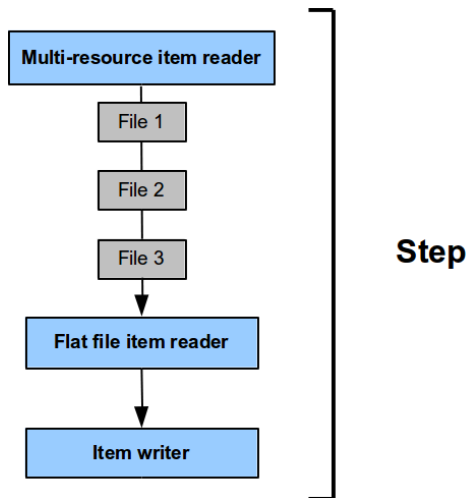
<task:executor id="taskExecutor" pool-size="5" />
```

Going further...

- ▶ Business delegation with an `ItemProcessor`
- ▶ Available `ItemProcessor` implementations
 - ▶ Adapter, validator
- ▶ The `ItemProcessor` can filter items

- ▶ Problem: I have multiple input files and I want to process them in parallel
- ▶ Solution: use partitioning to parallelize the processing on multiple threads

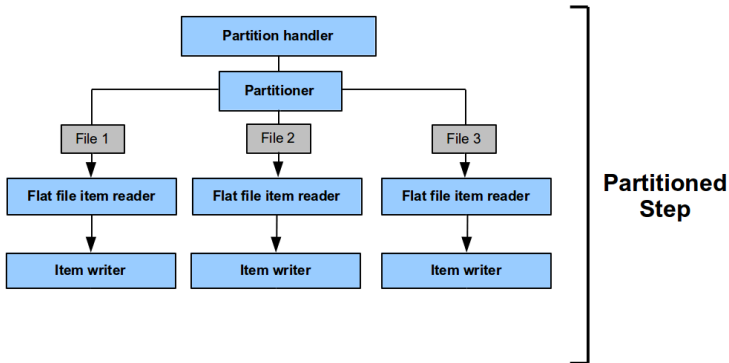
Serial processing



Partitioning in Spring Batch

- ▶ Partition the input data
 - ▶ e.g. one input file = one partition
 - ▶ partition processed in a dedicated step execution
- ▶ Partitioning is easy to set up but need some knowledge about the data
- ▶ Partition handler implementation
 - ▶ Multi-threaded
 - ▶ Spring Integration

Multi-threaded partitioning



Partitioner for input files

```
<bean id="partitioner"  
      class="o.s.b.core.partition.support.MultiResourcePartitioner">  
  <property name="resources"  
            value="file:./src/main/resources/input/*.txt" />  
</bean>
```

The partitioner sets a context for the components

```
<bean id="reader"
      class="org.springframework.batch.item.file.FlatFileItemReader"
      scope="step">
  (...)
  <property name="resource" value="#{stepExecutionContext['fileName']}" />
</bean>
```


Using the multi-threaded partition handler

```
<batch:job id="fileReadingPartitioningJob">
  <batch:step id="partitionedStep" >
    <batch:partition step="readWriteContactsPartitionedStep"
      partitioner="partitioner">
      <batch:handler task-executor="taskExecutor" />
    </batch:partition>
  </batch:step>
</batch:job>

<batch:step id="readWriteContactsPartitionedStep">
  <batch:tasklet>
    <batch:chunk reader="reader" writer="writer" commit-interval="10" />
  </batch:tasklet>
</batch:step>
```

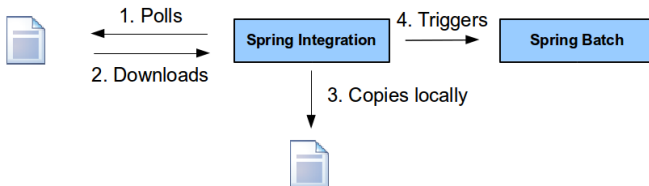
Going further...

- ▶ Spring Integration partition handler implementation
- ▶ Other scaling approaches
 - ▶ parallel steps, remote chunking, multi-threaded step)

- ▶ Problem: downloading files from a FTP server and processing them with Spring Batch
- ▶ Solution: use Spring Integration to poll the FTP server and trigger Spring Batch accordingly

Using Spring Integration for transfer and triggering

FTP Server



The launching code

```
public class FileContactJobLauncher {  
  
    public void launch(File file) throws Exception {  
        JobExecution exec = jobLauncher.run(  
            job,  
            new JobParametersBuilder()  
                .addString("input.file", "file:"+file.getAbsolutePath())  
                .toJobParameters()  
        );  
    }  
}
```

- The File is the local copy

Listening to the FTP server

```
<int:channel id="fileIn" />

<int-ftp:inbound-channel-adapter local-directory="file:./input"
    channel="fileIn" session-factory="ftpClientFactory"
    remote-directory="/" auto-create-local-directory="true">
    <int:poller fixed-rate="1000" />
</int-ftp:inbound-channel-adapter>

<bean id="ftpClientFactory"
    class="org.springframework.integration.ftp.session.DefaultFtpSessionFactory">
    <property name="host" value="localhost"/>
    <property name="port" value="2222"/>
    <property name="username" value="admin"/>
    <property name="password" value="admin"/>
</bean>
```

Calling the launcher on an inbound message

```
<int:channel id="fileIn" />

<int:service-activator input-channel="fileIn">
  <bean
    class="com.zenika.workshop.springbatch.integration.FileContactJobLauncher">
    <property name="job" ref="fileDroppingLaunchingJob" />
    <property name="jobLauncher" ref="jobLauncher" />
  </bean>
</int:service-activator>
```

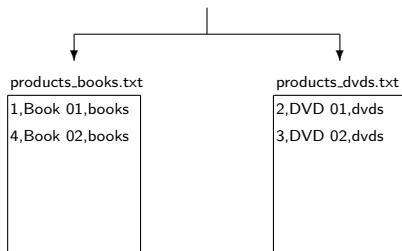
Going further...

- ▶ Checking Spring Integration connectors
 - ▶ Local file system, FTPS, SFTP, HTTP, JMS, etc.
- ▶ Checking operations on messages
 - ▶ Filtering, transforming, routing, etc.

- ▶ Problem: I want to export items from different categories from a database to files
- ▶ Solution: provide a partition strategy and use partitioning

The use case

ID	name	category
1	Book 01	books
2	DVD 01	dvds
3	DVD 02	dvds
4	Book 02	books



Partitioning based on categories

- ▶ 2 partitions in this case

ID	name	category
1	Book 01	books
2	DVD 01	dvds
3	DVD 02	dvds
4	Book 02	books

Partitioning logic with the Partitioner interface

```
public class ProductCategoryPartitioner implements Partitioner {  
    (...)  
  
    @Override  
    public Map<String, ExecutionContext> partition(int gridSize) {  
        List<String> categories = jdbcTemplate.queryForList(  
            "select distinct(category) from product",  
            String.class  
        );  
        Map<String, ExecutionContext> results =  
            new LinkedHashMap<String, ExecutionContext>();  
        for(String category : categories) {  
            ExecutionContext context = new ExecutionContext();  
            context.put("category", category);  
            results.put("partition."+category, context);  
        }  
        return results;  
    }  
}
```

Output of the Partitioner

► Excerpt:

```
for (String category : categories) {  
    ExecutionContext context = new ExecutionContext();  
    context.put("category", category);  
    results.put("partition."+category, context);  
}
```

► Results:

```
partition.books = { category => 'books' }  
partition.dvds  = { category => 'dvds' }
```

Components can refer to partition parameters

- ▶ They need to use the step scope

```
<bean id="reader"
      class="org.springframework.batch.item.database.JdbcCursorItemReader"
      scope="step">
  <property name="sql"
            value="select id,name,category from product where category = ?" />
  <property name="preparedStatementSetter">
    <bean class="org.springframework.jdbc.core.ArgPreparedStatementSetter">
      <constructor-arg value="#{stepExecutionContext['category']}" />
    </bean>
  </property>
</bean>

<bean id="writer"
      class="org.springframework.batch.item.file.FlatFileItemWriter"
      scope="step">
  <property name="resource"
            value="file:./target/products-#{stepExecutionContext['category']}.txt" />
  (...)
</bean>
```

Configure the partitioned step

- The default implementation is multi-threaded

```
<batch:job id="databaseReadingPartitioningJob">
  <batch:step id="partitionedStep" >
    <batch:partition step="readWriteProductsPartitionedStep"
      partitioner="partitioner">
      <batch:handler task-executor="taskExecutor" />
    </batch:partition>
  </batch:step>
</batch:job>

<batch:step id="readWriteProductsPartitionedStep">
  <batch:tasklet>
    <batch:chunk reader="reader" writer="writer" commit-interval="10" />
  </batch:tasklet>
</batch:step>
```

Going further...

- ▶ Check existing partitioner implementations
- ▶ Check other partition handler implementations
- ▶ Check other scaling strategies

- ▶ Problem: my job has a complex flow of steps, how can Spring Batch deal with it?
- ▶ Solution: use the step flow attributes and tags, as well as `StepExecutionListeners`.

The next attribute for a linear flow

```
<batch:job id="complexFlowJob">
  <batch:step id="digestStep" next="flatFileReadingStep">
    <batch:tasklet ref="digestTasklet" />
  </batch:step>
  <batch:step id="flatFileReadingStep">
    <batch:tasklet>
      <batch:chunk reader="reader" writer="writer" commit-interval="3" />
    </batch:tasklet>
  </batch:step>
</batch:job>
```

There's also a next tag non-linear flows

```
<batch:job id="complexFlowJob">
  <batch:step id="digestStep">
    <batch:tasklet ref="digestTasklet" />
    <batch:next on="COMPLETED" to="flatFileReadingStep"/>
    <batch:next on="FAILED" to="trackIncorrectFileStep"/>
  </batch:step>
  <batch:step id="flatFileReadingStep">
    <batch:tasklet>
      <batch:chunk reader="reader" writer="writer" commit-interval="3" />
    </batch:tasklet>
  </batch:step>
  <batch:step id="trackIncorrectFileStep">
    <batch:tasklet ref="trackIncorrectFileTasklet" />
  </batch:step>
</batch:job>
```

- NB: there are also the end, fail, and stop tags

But if I have more complex flows?

- ▶ The `next` attribute decision is based on the exit code
- ▶ You can use your own exit codes
- ▶ A `StepExecutionListener` can modify the exit code of a step execution

Modifying the exit code of a step execution

```
package com.zenika.workshop.springbatch;

import org.springframework.batch.core.ExitStatus;
import org.springframework.batch.core.StepExecution;
import org.springframework.batch.core.StepExecutionListener;

public class SkipsListener implements StepExecutionListener {

    @Override
    public void beforeStep(StepExecution stepExecution) { }

    @Override
    public ExitStatus afterStep(StepExecution stepExecution) {
        String exitCode = stepExecution.getExitStatus().getExitCode();
        if (!exitCode.equals(ExitStatus.FAILED.getExitCode()) &&
            stepExecution.getSkipCount() > 0) {
            return new ExitStatus("COMPLETED WITH SKIPS");
        } else {
            return null;
        }
    }
}
```

Plugging the SkipsListener

```
<step id="flatFileReadingStep">
  <tasklet>
    <chunk reader="reader" writer="writer"
      commit-interval="3" skip-limit="10">
      <skippable-exception-classes>
        <include
          class="org.springframework.batch.item.file.FlatFileParseException"/>
        </skippable-exception-classes>
      </chunk>
    </tasklet>
  <end on="COMPLETED"/>
  <next on="COMPLETED WITH SKIPS" to="trackSkipsStep"/>
  <listeners>
    <listener>
      <beans:bean class="com.zenika.workshop.springbatch.SkipsListener" />
    </listener>
  </listeners>
</step>
```

Going further...

- ▶ The `JobExecutionDecider` and the decision tag
- ▶ The stop, fail, and end tags

- ▶ Problem: I want to process an input file in an all-or-nothing manner.
- ▶ Solution: Don't do it. Atomic processing isn't batch processing! Anyway, if you really need an atomic processing, use a custom `CompletionPolicy`.

Why atomic processing is a bad idea?

- ▶ You loose the benefits of chunk-oriented processing
 - ▶ speed, small memory footprint, etc.
- ▶ On a rollback, you loose everything (that's perhaps the point!)
- ▶ The rollback can take a long time (several hours)
- ▶ It all depends on the amount of data and on the processing

I really need an atomic processing

- ▶ Rollback yourself, with compensating transactions
- ▶ Use a transaction rollback
 - ▶ It's only a never ending chunk!

Quick and dirty, large commit interval

- ▶ Set the commit interval to a very large value
- ▶ You should never have more items!

```
<batch:chunk reader="reader" writer="writer" commit-interval="1000000"/>
```

Use a never-ending CompletionPolicy

- ▶ Spring Batch uses a CompletionPolicy to know if a chunk is complete

```
package org.springframework.batch.repeat;  
  
public interface CompletionPolicy {  
    boolean isComplete(RepeatContext context, RepeatStatus result);  
    boolean isComplete(RepeatContext context);  
    RepeatContext start(RepeatContext parent);  
    void update(RepeatContext context);  
}
```

Plugging in the CompletionPolicy

```
<batch:job id="atomicProcessingJob">  
  <batch:step id="atomicProcessingStep">  
    <batch:tasklet>  
      <batch:chunk reader="reader" writer="writer"  
        chunk-completion-policy="atomicCompletionPolicy" />  
    </batch:tasklet>  
  </batch:step>  
</batch:job>
```

- ▶ NB: remove the `commit-interval` attribute when using a `CompletionStrategy`

Which CompletionPolicy for my atomic processing?

```
<bean id="atomicCompletionPolicy"  
      class="o.s.b.repeat.policy.DefaultResultCompletionPolicy" />
```

Going further...

- ▶ Flow in a job
- ▶ SkipPolicy, RetryPolicy