Spring Batch Workshop

Arnaud Cogoluègnes

Consultant with Zenika, Co-author Spring Batch in Action

May 24, 2011

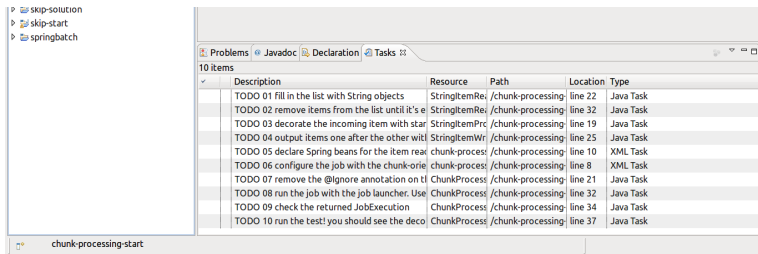## Overview

- This workshop highlights 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
- 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.

Follow the TODOs

- ▶ Track the TODO in the *-start projects!
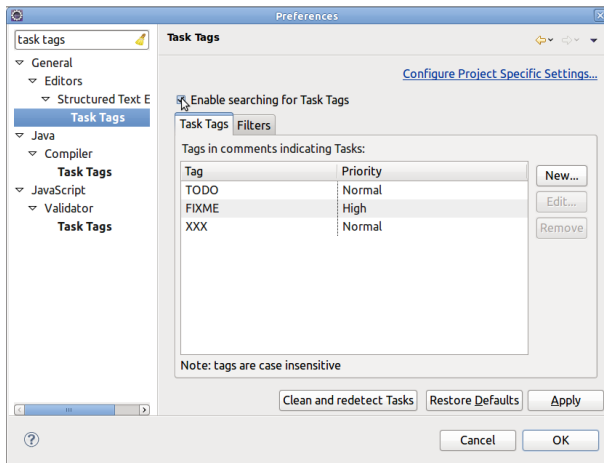- ▶ It's easier with support from the IDE

TODO with Eclipse

- Window > Preferences > "tasks tag" in filter

TODO with Eclipse

- Open the "Tasks" view
- click on the down arrow on the right
- "configure contents"
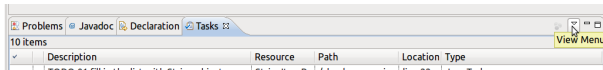
TODO with Eclipse

▶ Check "TODOs" on the left

▶ Check "On any element in the same project" on the right (scope)
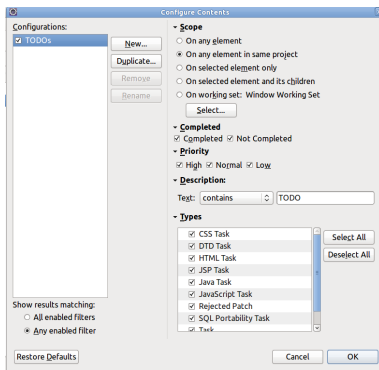
Spring support in IDE is a $+$

▶ e.g. code completion in SpringSource Tool Suite

Basic features for batch applications

- ► Read – process – write large amounts of data, efficiently
- ► Ready-to-use components to read from/write to
    - ► Flat/XML files
    - ► Databases (JDBC, Hibernate, JPA, iBatis)
    - ► JMS queues
    - ► Emails
- ► Numerous extension points/hooks

Advanced features for batch applications

- ► Configuration to skip/retry items
- ► Execution metadata
  - ► Monitoring
  - ► Restart after failure
- ► Scaling strategies
  - ► Local/remote
  - ► Partitioning, remote processing

- ▶ Problem: getting started with Spring Batch
- ▶ Solution: writing a simple "Hello World" job

Structure of a job

- ▶ A Spring Batch job is made of steps
- ▶ The Hello World job has one step
- ▶ The processing is implemented in a `Tasklet`

The Hello World `Tasklet`

```java
public class HelloWorldTasklet implements Tasklet {

  @Override
  public RepeatStatus execute(
      StepContribution contribution,
      ChunkContext chunkContext) throws Exception {
    System.out.println("Hello world!");
    return RepeatStatus.FINISHED;
  }

}
```

The configuration of the Hello World job [1]

```xml
<batch:job id="helloWorldJob">
  <batch:step id="helloWorldStep">
    <batch:tasklet>
      <bean class="com.zenika.workshop.springbatch.HelloWorldTasklet" />
    </batch:tasklet>
  </batch:step>
</batch:job>
```

---

[1] Notice the batch namespace

Spring Batch needs some infrastructure beans

► Let's use the typical test configuration

```
<bean id="transactionManager"
      class="o.s.b.support.transaction.ResourcelessTransactionManager" />

<bean id="jobRepository"
      class="o.s.b.core.repository.support.MapJobRepositoryFactoryBean" />

<bean id="jobLauncher"
      class="o.s.b.core.launch.support.SimpleJobLauncher">
  <property name="jobRepository" ref="jobRepository" />
</bean>
```

Running the test in a JUnit test

```
@RunWith( SpringJUnit4ClassRunner . class )
@ContextConfiguration ("/ hello-world-job . xml")
public class HelloWorldJobTest {

  @Autowired
  private Job job ;

  @Autowired
  private JobLauncher jobLauncher ;

  @Test public void helloWorld () throws Exception {
    JobExecution execution = jobLauncher . run ( job , new JobParameters ());
    assertEquals ( ExitStatus .COMPLETED, execution . getExitStatus ());
  }
}
```

- ▶ Problem: processing large amounts of data efficiently
- ▶ Solution: using chunk processing

What is chunk processing?

- ▶ Batch jobs often read, process, and write items
- ▶ e.g.
  - ▶ Reading items from a file
  - ▶ Then processing (converting) items
  - ▶ Writing items to a database
- ▶ Spring Batch calls this "chunk processing"
- ▶ a chunk = a set of items

Chunk processing with Spring Batch

- Spring Batch
    - handles the iteration logic
    - uses a transaction for each chunk
    - lets you choose the chunk size
    - defines interfaces for each part of the processing

The reading phase

▶ Spring Batch creates chunks of items by calling read()

▶ Reading ends when read() returns null

```
public interface ItemReader<T> {

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

}
```

The processing phase

- Once a chunk is created, items are sent to the processor
- Optional

```
public interface ItemProcessor<I, O> {

  O process(I item) throws Exception;

}
```

The writing phase

- ▶ Receives all the items of the chunk
- ▶ Allows for batch update (more efficient)

```java
public interface ItemWriter<T> {

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

}
```

An example

- ▶ Let's implement a (too?) simple chunk-oriented step!

The ItemReader

```java
public class StringItemReader implements ItemReader<String> {

  private List<String> list;

  public StringItemReader() {
    this.list = new ArrayList<String>(Arrays.asList(
      "1","2","3","4","5","6","7")
    );
  }

  @Override
  public String read() throws Exception, UnexpectedInputException,
                        ParseException, NonTransientResourceException {
    return !list.isEmpty() ? list.remove(0) : null;
  }
}
```

The ItemProcessor

```java
public class StringItemProcessor implements ItemProcessor<String, String> {

  @Override
  public String process(String item) throws Exception {
    return "*** "+item+" ***";
  }

}
```

The ItemWriter

```java
public class StringItemWriter implements ItemWriter<String> {

  private static final Logger LOGGER =
    LoggerFactory.getLogger(StringItemWriter.class);

  @Override
  public void write(List<? extends String> items) throws Exception {
    for(String item : items) {
      LOGGER.info("writing "+item);
    }
  }

}
```

Configuring the job

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

<bean id="reader" class="com.zenika.workshop.springbatch.StringItemReader" />

<bean id="processor"
      class="com.zenika.workshop.springbatch.StringItemProcessor" />

<bean id="writer" class="com.zenika.workshop.springbatch.StringItemWriter" />
```

Considerations

- Do I always need to write my
  ItemReader/Processor/Writer?
- No, Spring Batch provides ready-to-use components for
  common datastores
    - Flat/XML files, databases, JMS, etc.
- As an application developer, you
    - Configure these components
    - Provides some logic (e.g. mapping a line with a domain object)

Going further...

- ▶ Reader/writer implementation for flat/XML files, database, JMS
- ▶ Skipping items when something goes wrong
- ▶ Listeners to react to the chunk processing

- ▶ Problem: reading lines from a flat file and sending them to another source (e.g. database)
- ▶ Solution: using the FlatFileItemReader

Spring Batch's support for flat file reading

- ▶ Spring Batch has built-in support for flat files
  - ▶ Through the FlatFileItemReader for reading
- ▶ The FlatFileItemReader handles I/O
- ▶ 2 main steps:
  - ▶ Configuring the FlatFileItemReader
  - ▶ Providing a line-to-object mapping strategy

The usual suspects

```
Susy , Hauerstock ,2010−03−04
De Anna , Raghunath ,2010−03−04
Kiam , Whitehurst ,2010−03−04
Alecia , Van Holst ,2010−03−04
Hing , Senecal ,2010−03−04
```

```
public class Contact {

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

  (...)
}
```

What do we need to read a flat file?

- ▶ How to tokenize a line
- ▶ How to map the line with a Java object
- ▶ Where to find the file to read

The FlatFileItemReader configuration

```xml
<bean id="reader"
      class="org.springframework.batch.item.file.FlatFileItemReader">
  <property name="lineMapper">
    <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
      <property name="lineTokenizer">
        <bean
  class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
          <property name="names" value="firstname,lastname,birth" />
        </bean>
      </property>
      <property name="fieldSetMapper">
        <bean class="com.zenika.workshop.springbatch.ContactFieldSetMapper" />
      </property>
    </bean>
  </property>
  <property name="resource" value="classpath:contacts.txt" />
</bean>
```

The line-to-object mapping strategy

- ▶ A `FieldSetMapper` to map a line with an object
- ▶ More about business logic, so typically implemented by developer
- ▶ Spring Batch provides straightforward implementations

Custom `FieldSetMapper` implementation

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

import org.springframework.batch.item.file.mapping.FieldSetMapper;
import org.springframework.batch.item.file.transform.FieldSet;
import org.springframework.validation.BindException;

public class ContactFieldSetMapper implements FieldSetMapper<Contact> {

  @Override
  public Contact mapFieldSet(FieldSet fieldSet) throws BindException {
    return new Contact(
      fieldSet.readString("firstname"),
      fieldSet.readString("lastname"),
      fieldSet.readDate("birth","yyyy-MM-dd")
    );
  }
}
```

Going further...

- ▶ `FlatFileItemWriter` to write flat file
- ▶ Fixed-length format (different tokenizer)
- ▶ Skipping badly formatted lines

- ▶ Problem: my job fails miserably because of a tiny error in my input file
- ▶ Solution: skipping lines without failing the whole execution

A CSV file with a badly formatted line

```
Susy , Hauerstock ,2010−03−04
De−Anna , Raghunath ,2010−03−04
Kiam , Whitehurst ,2010−03−04
Alecia , Van Holst ,09-23-2010
Hing , Senecal ,2010−03−04
Kannan , Pirkle ,2010−03−04
Row , Maudrie ,2010−03−04
Voort , Philbeck ,2010−03−04
```

Skip configuration

- ▶ Choose the exceptions to skip
- ▶ Set the max number of items to skip

```xml
<batch:job id="skipJob">
  <batch:step id="skipStep">
    <batch:tasklet>
      <batch:chunk reader="reader" writer="writer" commit-interval="3"
                   skip-limit="10">
        <batch:skippable-exception-classes>
          <batch:include
           class="org.springframework.batch.item.file.FlatFileParseException"/>
        </batch:skippable-exception-classes>
      </batch:chunk>
    </batch:tasklet>
  </batch:step>
</batch:job>
```

Going further...

- Logging skipped items with a `SkipListener`
- Setting a custom `SkipPolicy`

- ▶ Problem: passing values to the configuration when launching a job
- ▶ Solution: using job parameters and late binding

Use case: providing a input file dynamically to the item reader

```
JobParameters jobParameters = new JobParametersBuilder()
    .addString("input.file", "file:./input/contacts-01.txt")
    .toJobParameters();
JobExecution execution = jobLauncher.run(job, jobParameters);
```

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

Going further...

- ▶ Spring Expression Language (SpEL)
- ▶ Step scope for partitioning

- ▶ Problem: reading large result sets from the database with a stable memory footprint
- ▶ Solution: using the `JdbcPagingItemReader`, which uses paging to handle large result sets

JdbcPagingItemReader configuration

```xml
<bean id="reader"
      class="org.springframework.batch.item.database.JdbcPagingItemReader">
  <property name="dataSource" ref="dataSource" />
  <property name="pageSize" value="10" />
  <property name="queryProvider">
    <bean class="o.s.b.item.database.support.SqlPagingQueryProviderFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="selectClause"
                value="select id,firstname,lastname,birth" />
      <property name="fromClause" value="from contact" />
      <property name="sortKey" value="id" />
    </bean>
  </property>
  <property name="rowMapper">
    <bean class="com.zenika.workshop.springbatch.ContactRowMapper" />
  </property>
</bean>
```

Paging or cursors?

- ▶ By paging, you send multiple queries to the database
- ▶ Alternative: cursor-based item reader
  - ▶ Spring Batch "streams" the result set from the DB
  - ▶ Only one query
- ▶ Paging always works, cursor-based reader depends on driver implementation and database

Going further...

- ▶ Paging readers for Hibernate, JPA, iBatis
- ▶ Cursor-based readers

- ▶ Problem: monitoring the execution of batch jobs
- ▶ Solution: letting Spring Batch storing execution metadata in a database

Why storing execution metadata?

- ▶ Spring Batch keeps track of batch execution
- ▶ Enables:
    - ▶ Monitoring by querying metadata tables
    - ▶ Restarting after a failure

Job, job instance, and job execution
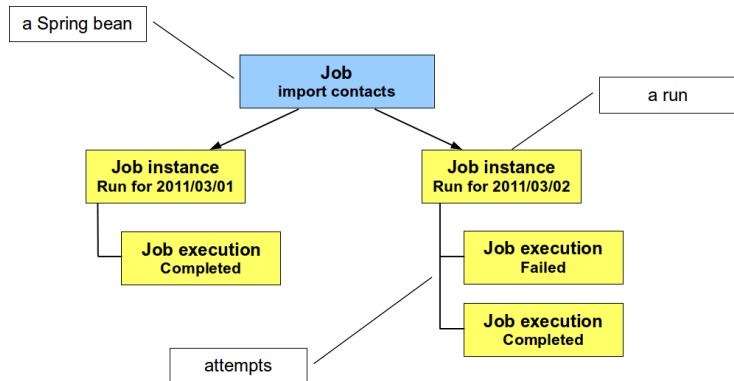
Job instance

- ▶ How to define a job instance?
- ▶ Thanks to job parameters
- ▶ Job parameters define the identity of the job instance

The reading phase

- ▶ Metadata are stored in a database
  - ▶ In-memory implementation for test/development
- ▶ Monitoring tools can query metadata tables

Going further...

- ▶ Spring Batch Admin, the web console for Spring Batch
- ▶ `JobExplorer` and `JobOperator` interfaces
- ▶ Spring JMX support

- Problem: scheduling a job to execute periodically
- Solution: using the scheduling support in Spring

A class to launch the job

```java
public class ImportLauncher {

  public void launch() throws Exception {
    JobExecution exec = jobLauncher.run(
      job,
      new JobParametersBuilder()
        .addLong("time", System.currentTimeMillis())
        .toJobParameters()
    );
  }
}
```

Spring scheduling configuration

```
<bean id="importLauncher"
      class="com.zenika.workshop.springbatch.ImportLauncher" />

<task:scheduled-tasks>
  <task:scheduled ref="importLauncher" method="launch"
                  fixed-delay="1000" />
</task:scheduled-tasks>
```

▶ `cron` attribute available

Going further...

- ▶ Threading settings in Spring Scheduler
- ▶ Spring support for Quartz

- ▶ Problem: I want to add some business logic before writing the items I just read
- ▶ Solution: use an `ItemProcessor` to process/convert read items before sending them to the `ItemWriter`

Use case

- Reading contacts from a flat file
- Registering them into the system
    - This is the *business logic*
- Writing the registration confirmations to the database

The ItemProcessor interface

```java
public interface ItemProcessor<I, O> {

  O process(I item) throws Exception;

}
```

How to implement an `ItemProcessor`

▶ An `ItemProcessor` usually delegates to existing business code

```java
public class ContactItemProcessor implements
              ItemProcessor<Contact, RegistrationConfirmation> {

  private RegistrationService registrationService;

  @Override
  public RegistrationConfirmation process(Contact item)
       throws Exception {
    return registrationService.process(item);
  }

}
```

Registering the `ItemProcessor`

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

<bean id="registrationService"
      class="com.zenika.workshop.springbatch.RegistrationService" />

<bean id="processor"
      class="com.zenika.workshop.springbatch.ContactItemProcessor">
  <property name="registrationService" ref="registrationService" />
</bean>
```

Going further...

- ▶ Available `ItemProcessor` implementations
    - ▶ Adapter, validator
- ▶ The `ItemProcessor` can filter items

- ▶ Problem: logging skipped items
- ▶ Solution: using a `SkipListener`

2 steps to log skipped items

- ▶ Writing the `SkipListener` (and the logging code)
- ▶ Registering the listener on the step

Writing the SkipListener

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

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.core.listener.SkipListenerSupport;

public class Slf4jSkipListener<T,S> extends SkipListenerSupport<T, S> {

  private static final Logger LOG = LoggerFactory.getLogger(
      Slf4jSkipListener.class);

  @Override
  public void onSkipInRead(Throwable t) {
    LOG.warn("skipped item: {}",t.toString());
  }

}
```

Registering the SkipListener

```xml
<batch:job id="loggingSkippedItemsJob">
  <batch:step id="loggingSkippedItemsStep">
    <batch:tasklet>
      <batch:chunk reader="reader" writer="writer" commit-interval="3"
                   skip-limit="10">
        <batch:skippable-exception-classes>
          <batch:include
           class="org.springframework.batch.item.file.FlatFileParseException"/>
        </batch:skippable-exception-classes>
      </batch:chunk>
      <batch:listeners>
        <batch:listener ref="skipListener" />
      </batch:listeners>
    </batch:tasklet>
  </batch:step>
</batch:job>

<bean id="skipListener" class="com.zenika.workshop.springbatch.Slf4jSkipListener" />
```
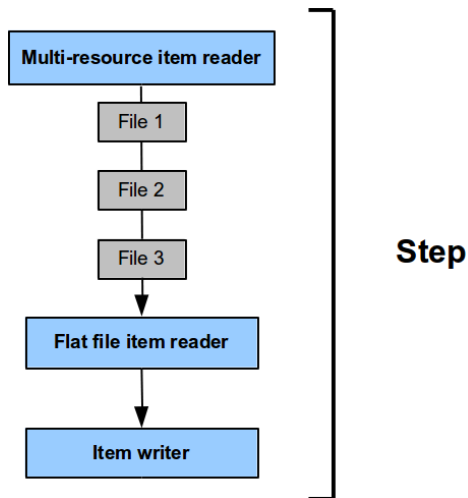
Going further...

- ▶ Other listeners in Spring Batch
  - ▶ ChunkListener, Item(Read/Process/Write)Listener, ItemStream, StepExecutionListener, JobExecutionListener

- ▶ 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
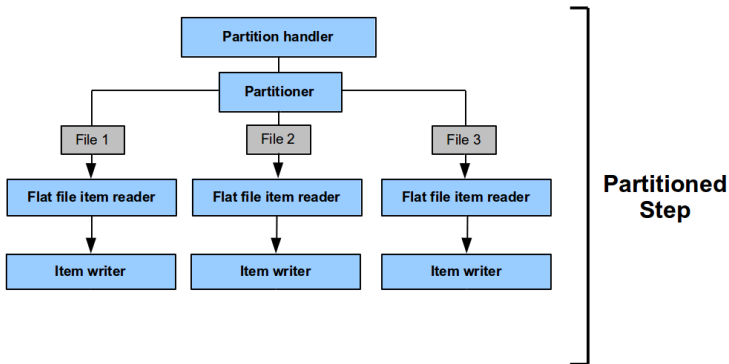
Partitioner for input files

```
<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
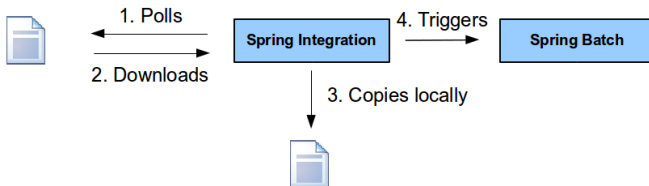
The launching code

```java
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="com.zenika.workshop.springbatch.integration.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.