

# Spring Batch Workshop

Arnaud Cogoluègnes  
Consultant at Zenika, co-author “Spring Batch in Action”

# 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>

# Settings

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

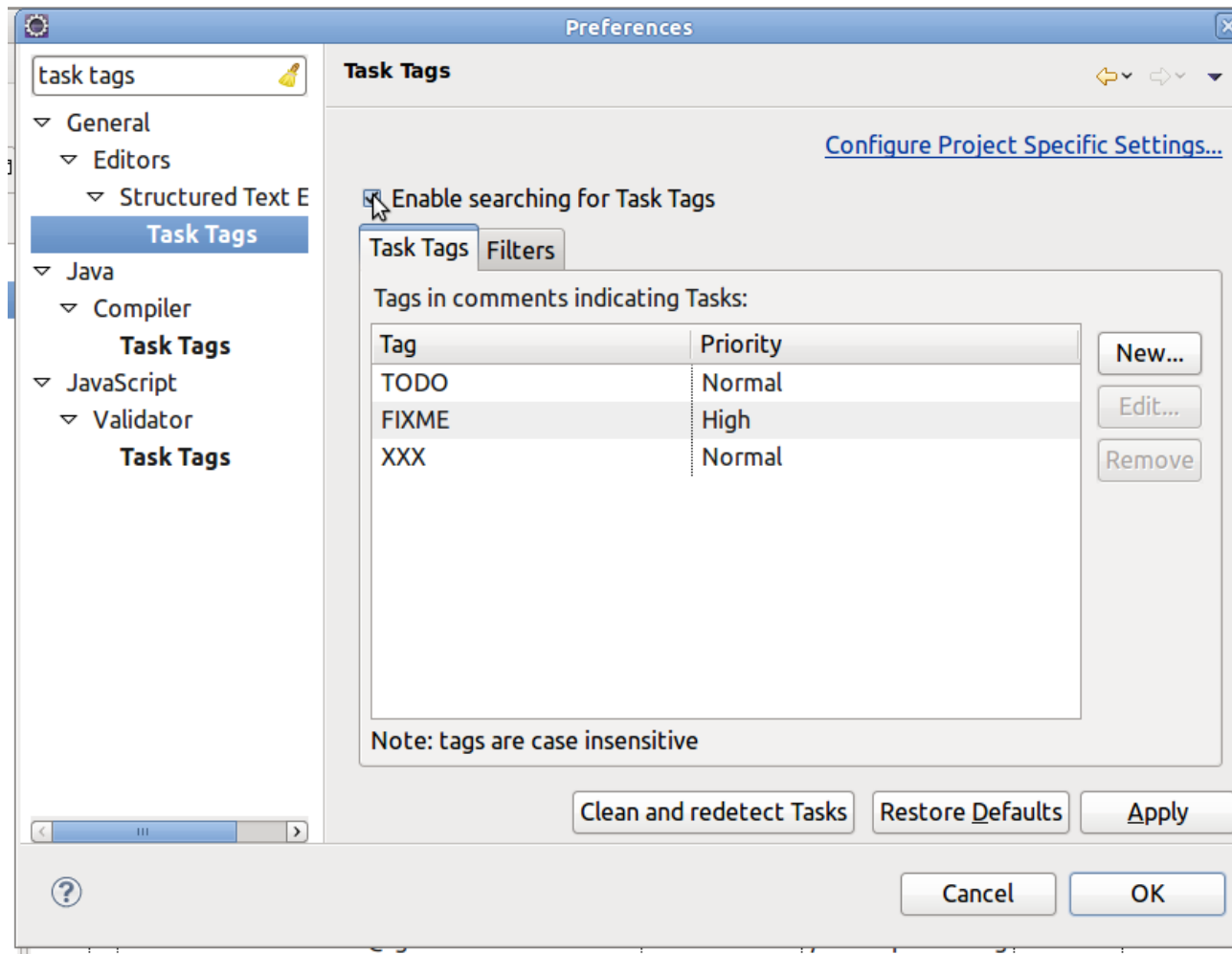
The screenshot displays an IDE interface. On the left, a project explorer shows a tree structure with the following items: `hello-world-start`, `skip-solution`, `skip-start`, and `springbatch`. The main editor area is divided into tabs: `Problems`, `Javadoc`, `Declaration`, and `Tasks`. The `Tasks` tab is active, showing a list of 10 items. The list is organized into columns: `Description`, `Resource`, `Path`, `Location`, and `Type`. The items are as follows:

| Description                                     | Resource       | Path               | Location | Type      |
|---|----------------|--------------------|----------|-----------|
| TODO 01 fill in the list with String objects    | StringItemRea  | /chunk-processing- | line 22  | Java Task |
| TODO 02 remove items from the list until it's e | StringItemRea  | /chunk-processing- | line 32  | Java Task |
| TODO 03 decorate the incoming item with star    | StringItemProc | /chunk-processing- | line 19  | Java Task |
| TODO 04 output items one after the other with   | StringItemWr   | /chunk-processing- | line 25  | Java Task |
| TODO 05 declare Spring beans for the item read  | chunk-proces   | /chunk-processing- | line 10  | XML Task  |
| TODO 06 configure the job with the chunk-orie   | chunk-proces   | /chunk-processing- | line 8   | XML Task  |
| TODO 07 remove the @Ignore annotation on t      | ChunkProcess   | /chunk-processing- | line 21  | Java Task |
| TODO 08 run the job with the job launcher. Use  | ChunkProcess   | /chunk-processing- | line 32  | Java Task |
| TODO 09 check the returned JobExecution         | ChunkProcess   | /chunk-processing- | line 34  | Java Task |
| TODO 10 run the test! you should see the deco   | ChunkProcess   | /chunk-processing- | line 37  | Java Task |

The bottom status bar of the IDE shows the text `chunk-processing-start`.

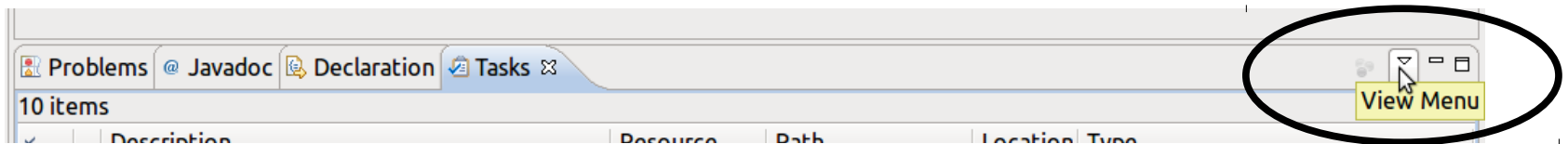
# TODO with Eclipse

- Window > Preferences > “tasks tag” in filter

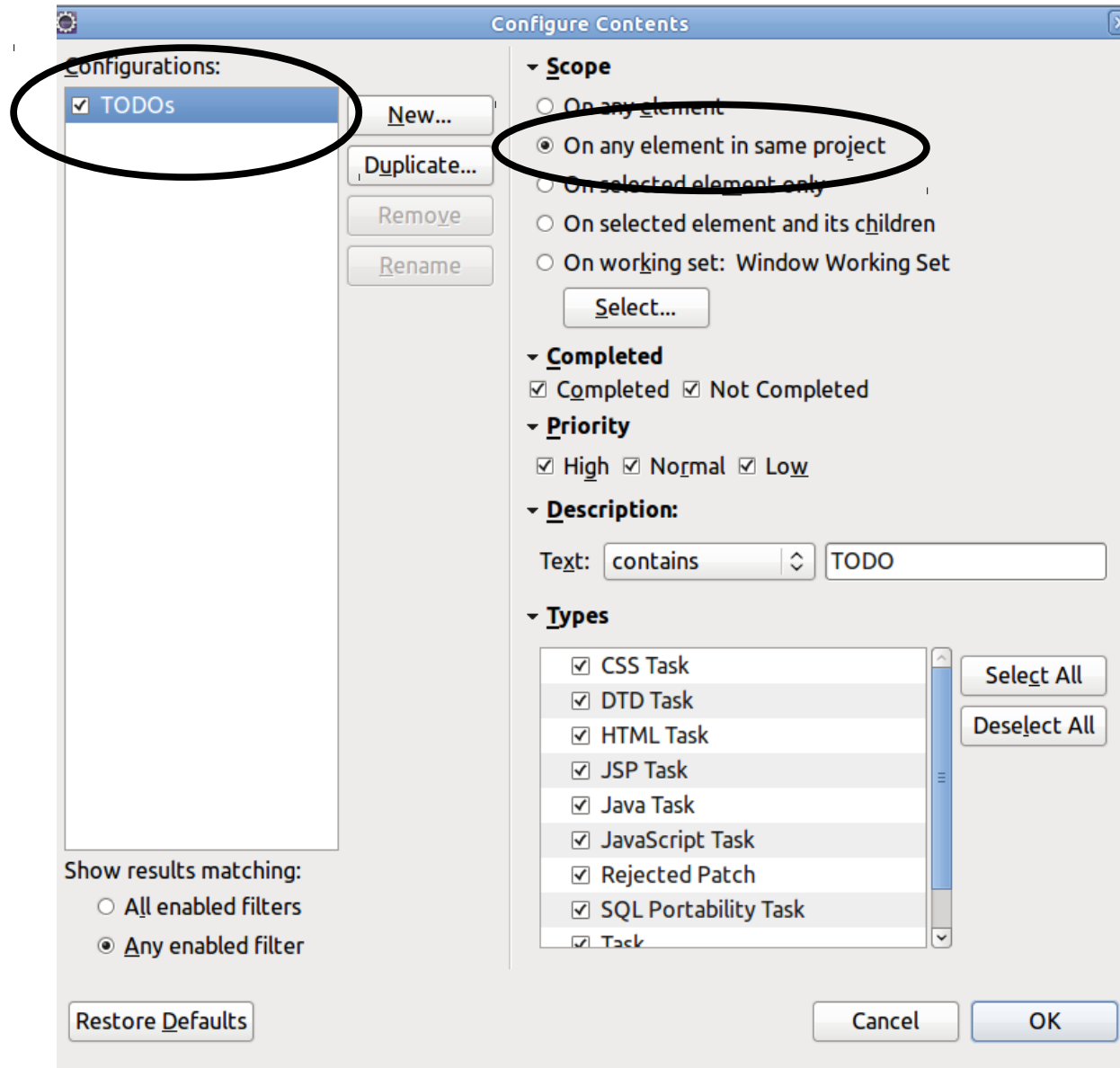


# TODO with Eclipse

- Open the “Tasks” view and “configure contents”

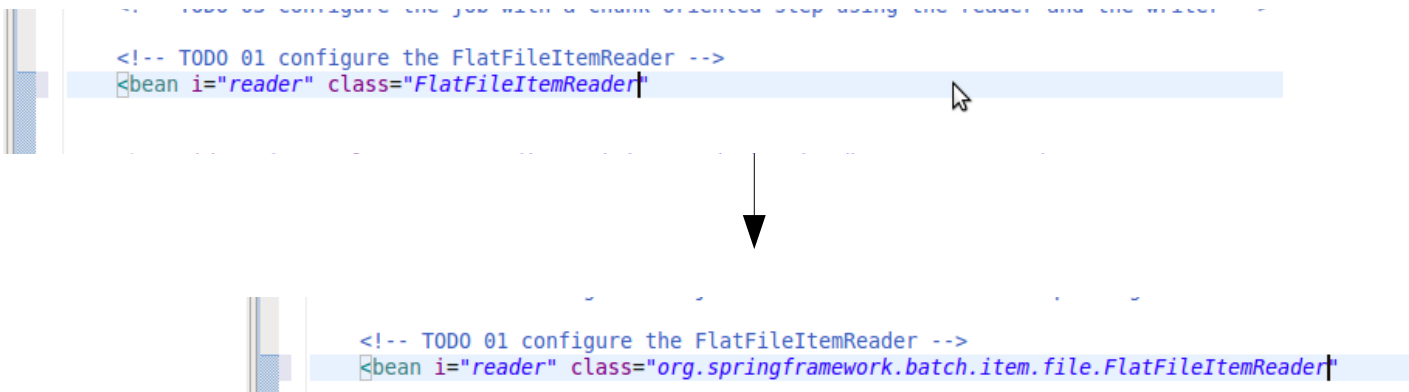


# TODO with Eclipse



# Spring support in IDE

- Spring support in IDE is a +
- e.g. code completion in SpringSource Tool Suite



# Spring Batch overview

- 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



# Spring Batch overview

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

# Hello World

- Problem: getting started with Spring Batch
- Solution: writing a simple “Hello world” job

# Hello World

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

# Hello World

- The Hello World Tasklet

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

# Hello World

- The configuration of the Hello World job
  - Notice the `<batch />` namespace

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

# Hello World

- 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>
```

# Hello World

- Let's 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());
    }
}
```

# Chunk processing

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



# 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

- 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

# Chunk processing

- ItemReader
  - Reading ends when read() returns null

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

# Chunk processing

- ItemProcessor
  - optional

```
public interface ItemProcessor<I, O> {  
    O process(I item) throws Exception;  
}
```

# Chunk processing

- ItemWriting
  - Receive all the items of the chunk
  - Allows for batch update (more efficient)

```
public interface ItemWriter<T> {  
    void write(List<? extends T> items) throws Exception;  
}
```

# Chunk processing

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

# Chunk processing

- The ItemReader

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

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.springframework.batch.item.ItemReader;
import org.springframework.batch.item.NonTransientResourceException;
import org.springframework.batch.item.ParseException;
import org.springframework.batch.item.UnexpectedInputException;

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;
    }
}
```

# Chunk processing

- The ItemProcessor

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

import org.springframework.batch.item.ItemProcessor;

public class StringItemProcessor implements ItemProcessor<String, String> {

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

}
```



# Chunk processing

- The ItemWriter

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

import java.util.List;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.item.ItemWriter;

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);
        }
    }
}
```

# Chunk processing

- 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" />
```

# Chunk processing

- 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.
- You
  - Configure these components
  - Provides some logic
    - e.g. mapping a line with a domain object

# Chunk processing

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

# Flat file reading

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


# 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 – object mapping strategy

# Flat file reading

- The usual suspects:

```
De-Anna,Raghunath,2010-03-04  
Susy,Hauerstock,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;  
  
    (...)  
}
```

# Flat file reading

- 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



# Flat file reading

Tokenization

```
<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.csv" />
  </bean>
```

File to read

Line – object mapping

# Flat file reading

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

# Flat file reading

```
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")
        );
    }
}
```

# Flat file reading

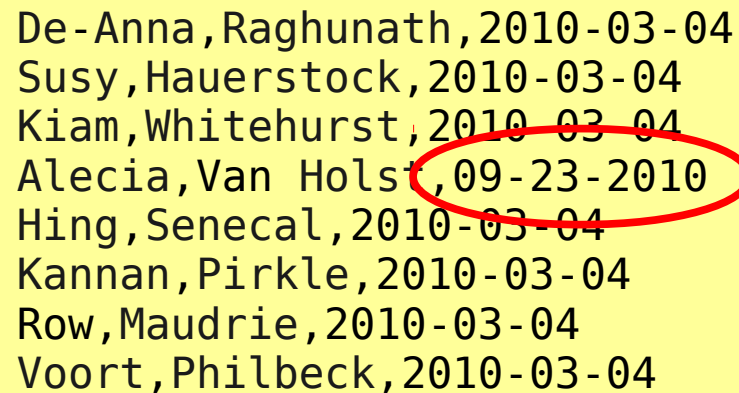
- Going further...
  - FlatFileItemWriter to write flat file
  - Fixed-length format (different tokenizer)
  - Skipping badly formatted lines

# Skip

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

# Skip

- Skipping lines is sometimes acceptable



```
De-Anna,Raghunath,2010-03-04  
Susy,Hauerstock,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

- Skip in Spring Batch
  - Choose the exceptions to skip
  - Set the max number of items to skip

```
<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>
```

# Skip

- Going further...
  - Logging skipped items with a SkipListener
  - Setting a custom SkipPolicy



# Dynamic job parameters

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

# Dynamic job parameters

- 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>
```

# Dynamic job parameters

- Going further...
  - Checking other available variables in an expression

# JDBC paging

- 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

# JDBC paging

```
<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>
```

# JDBC paging

- 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

# JDBC paging

- Going further...
  - Paging readers for Hibernate, JPA, iBatis
  - Cursor-based readers

# Execution metadata

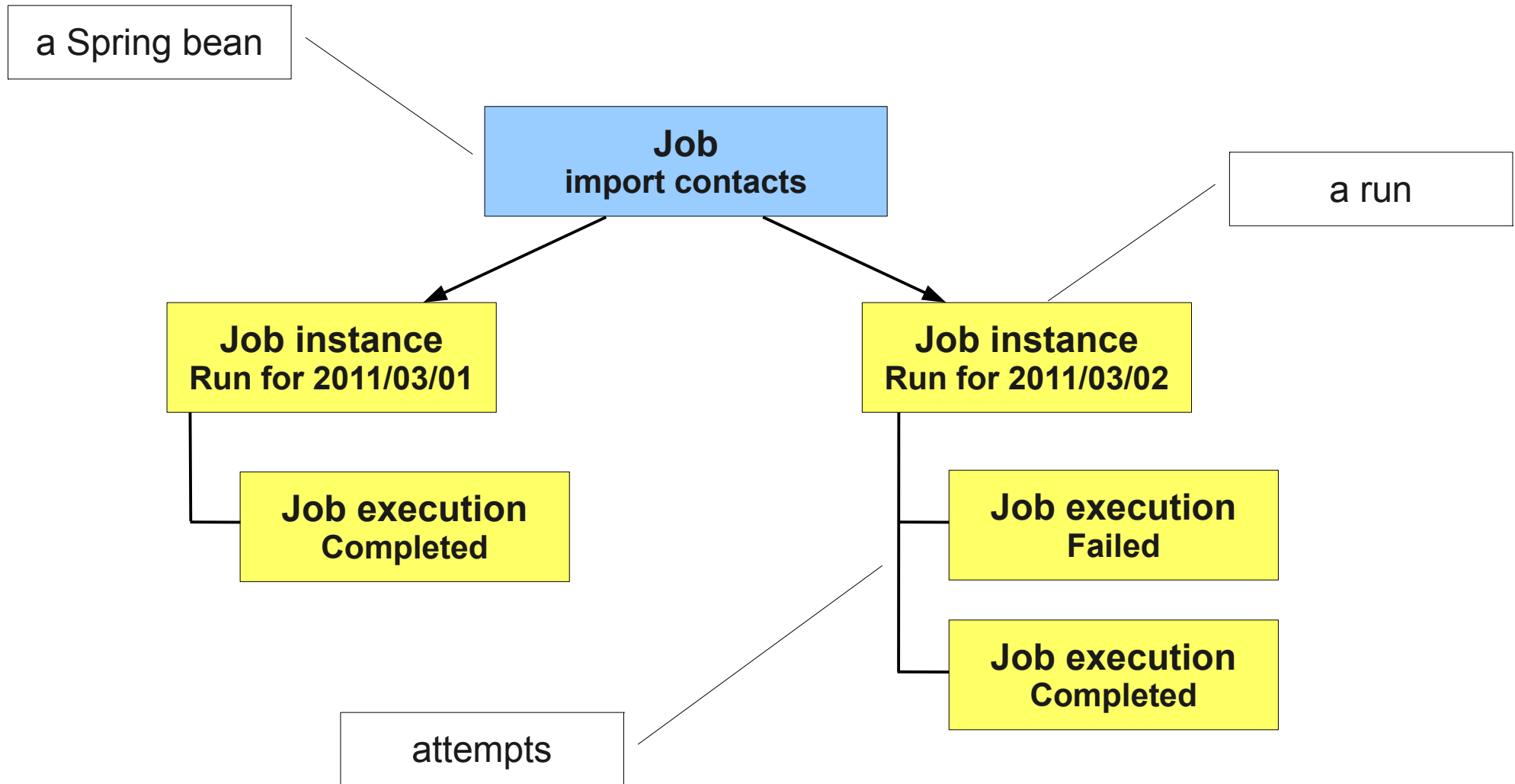
- Problem: monitoring the execution of batch jobs
- Solution: letting Spring Batch storing execution metadata in a database



# Execution metadata

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

# Execution metadata



# Execution metadata

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

# Execution metadata

- Metadata are stored in a database
  - In-memory implementation for test/development
- Monitoring tools can query metadata tables
  - e.g. Spring Batch Admin

# Execution metadata

- Going further...
  - Spring Batch Admin set-up
  - JobExplorer and JobOperator interfaces
  - Spring JMX support

# Scheduling

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

# Scheduling

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

```
<bean id="importLauncher"  
      class="com.zenika.workshop.springbatch.ImportLauncher" />  
  
<task:scheduled-tasks>  
    <task:scheduled ref="importLauncher" method="launch"  
                    fixed-delay="1000" />  
</task:scheduled-tasks>
```

A "cron" attribute is available

# Scheduling

- Going further...
  - Threading settings in Spring Scheduler
  - Spring support for Quartz



# Item processor

- Problem: I want to add some business logic before writing the items I just read
- Solution: use an item processor to process/convert read items before sending them to the item writer

# Item processor

- Use case:
  - Reading contacts from a flat file
  - Registering them into the system
  - Writing the registration confirmations to the database



Business logic

# Item processor

```
public interface ItemProcessor<I, O> {  
    O process(I item) throws Exception;  
}
```

# Item processor

- Delegate to business service

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

    private RegistrationService registrationService;

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

# Item processor

- Register the item processor on the step

```
<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>
```

# Item processor

- Going further...
  - Available ItemProcessor implementations

# Logging skipped items

- Problem: logging skipped items
- Solution: using a skip listener

# Logging skipped items

- 2 steps:
  - Writing the skip listener (and the logging code)
  - Registering the listener on the step



# Logging skipped items

- Writing the skip listener

```
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());
    }
}
```

# Logging skipped items

- Registering the skip listener

```
<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" />
```

# Logging skipped items

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

# File reading partitioning

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

# File reading partitioning

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

# File reading 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>
```

- Set a context for the steps to run

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

# File reading partitioning

- 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>
```