

Spring Batch Workshop

Arnaud Cogoluègnes

Consultant with Zenika, Co-author Spring Batch in Action

May 23, 2011

Overview

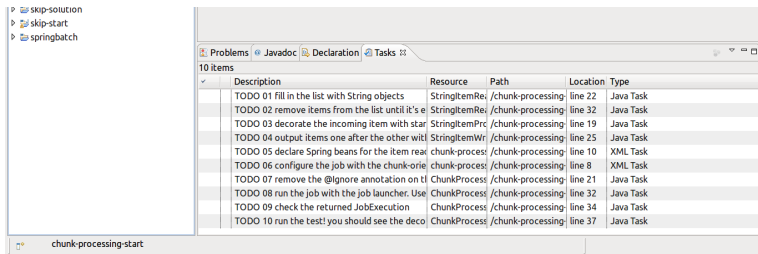
- ▶ This workshop highlights Spring Batch features
- ▶ Problem/solution approach
 - ▶ A few slides to cover the feature
 - ▶ A project to start from, just follow the TODOs
- ▶ Prerequisites
 - ▶ Basics about Java and Java EE
 - ▶ Spring: dependency injection, enterprise support
- ▶ <https://github.com/acogoluegnes/Spring-Batch-Workshop>

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

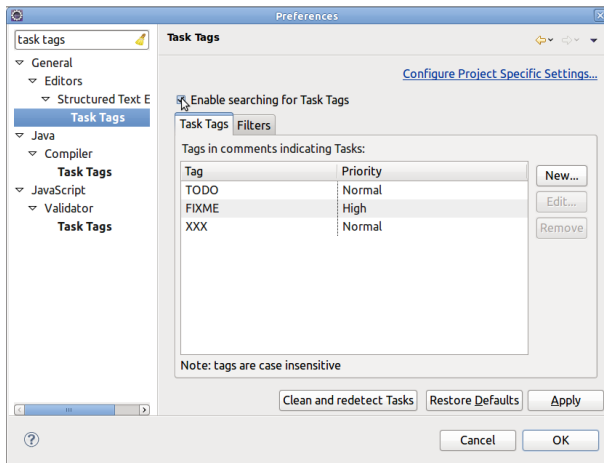


The screenshot shows an IDE interface with a project explorer on the left containing 'skip-solution', 'skip-start', and 'springbatch'. The main window displays the 'Tasks' tab for the 'chunk-processing-start' project, listing 10 TODO items. The tasks are organized into a table with columns for Description, Resource, Path, Location, and Type.

Description	Resource	Path	Location	Type
TODO 01 fill in the list with String objects	StringItemRe	/chunk-processing	line 22	Java Task
TODO 02 remove items from the list until it's e	StringItemRe	/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-ori	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

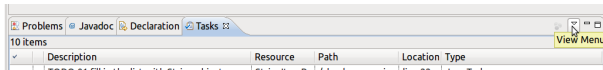
TODO with Eclipse

- ▶ Window > Preferences > “tasks tag” in filter



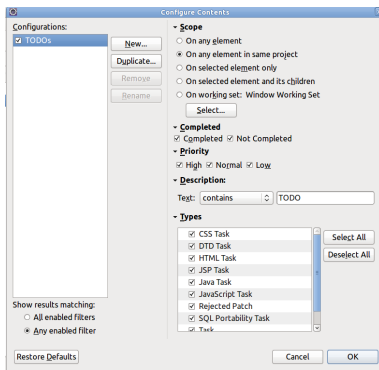
TODO with Eclipse

- ▶ 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

```
<!-- TODO 03 configure the job with a chunk-oriented step using the reader and the writer -->  
  
<!-- TODO 01 configure the FlatFileItemReader -->  
<bean id="reader" class="FlatFileItemReader"
```



```
<!-- TODO 03 configure the job with a chunk-oriented step using the reader and the writer -->  
  
<!-- TODO 01 configure the FlatFileItemReader -->  
<bean id="reader" class="org.springframework.batch.item.file.FlatFileItemReader"
```


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

```
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

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

¹Notice 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)

```
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

```
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

```
public class StringItemProcessor implements ItemProcessor<String, String> {  
  
    @Override  
    public String process(String item) throws Exception {  
        return "*** "+item+" ***";  
    }  
}
```

The 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);  
        }  
    }  
}
```

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

```
<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

```
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

```
<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