Spring Batch Workshop

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

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

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