Table of Contents

Chapter 1. What's New in Spring Batch 4.1

The Spring Batch 4.1 release adds the following features:

- A new `@SpringBatchTest` annotation to simplify testing batch components
- A new `@EnableBatchIntegration` annotation to simplify remote chunking and partitioning configuration
- A new `JsonItemReader` and `JsonFileItemWriter` to support the JSON format
- Add support for validating items with the Bean Validation API
- Add support for JSR-305 annotations
- Enhancements to the `FlatFileItemWriterBuilder` API

1.1. `@SpringBatchTest` Annotation

Spring Batch provides some nice utility classes (such as the `JobLauncherTestUtils` and `JobRepositoryTestUtils`) and test execution listeners (`StepScopeTestExecutionListener` and `JobScopeTestExecutionListener`) to test batch components. However, in order to use these utilities, you must configure them explicitly. This release introduces a new annotation named `@SpringBatchTest` that automatically adds utility beans and listeners to the test context and makes them available for autowiring, as the following example shows:

```
@RunWith(SpringRunner.class)
@SpringBatchTest
@ContextConfiguration(classes = {JobConfiguration.class})
public class JobTest {

   @Autowired
   private JobLauncherTestUtils jobLauncherTestUtils;

   @Autowired
   private JobRepositoryTestUtils jobRepositoryTestUtils;


   @Before
   public void clearMetadata() {
      jobRepositoryTestUtils.removeJobExecutions();
   }

   @Test
   public void testJob() throws Exception {
      // given
      JobParameters jobParameters =
            jobLauncherTestUtils.getUniqueJobParameters();

      // when
      JobExecution jobExecution =
            jobLauncherTestUtils.launchJob(jobParameters);

      // then
      Assert.assertEquals(ExitStatus.COMPLETED,
                          jobExecution.getExitStatus());
   }

}
```

For more details about this new annotation, see the Unit Testing chapter.

1.2. `@EnableBatchIntegration` Annotation

Setting up a remote chunking job requires the definition of a number of beans:

- A connection factory to acquire connections from the messaging middleware (JMS, AMQP, and others)
- A `MessagingTemplate` to send requests from the master to the workers and back again
- An input channel and an output channel for Spring Integration to get messages from the messaging middleware
- A special item writer (`ChunkMessageChannelItemWriter`) on the master side that knows how to send chunks of data to workers for processing and writing

- A message listener (`ChunkProcessorChunkHandler`) on the worker side to receive data from the master

This can be a bit daunting at first glance. This release introduces a new annotation named `@EnableBatchIntegration` as well as new APIs (`RemoteChunkingMasterStepBuilder` and `RemoteChunkingWorkerBuilder`) to simplify the configuration. The following example shows how to use the new annotation and APIs:

```
@Configuration
@EnableBatchProcessing
@EnableBatchIntegration
public class RemoteChunkingAppConfig {

   @Autowired
   private RemoteChunkingMasterStepBuilderFactory masterStepBuilderFactory;

   @Autowired
   private RemoteChunkingWorkerBuilder workerBuilder;

   @Bean
   public TaskletStep masterStep() {
         return this.masterStepBuilderFactory
                     .get("masterStep")
                     .chunk(100)
                     .reader(itemReader())
                     .outputChannel(outgoingRequestsToWorkers())
                     .inputChannel(incomingRepliesFromWorkers())
                     .build();
   }

   @Bean
   public IntegrationFlow worker() {
         return this.workerBuilder
                     .itemProcessor(itemProcessor())
                     .itemWriter(itemWriter())
                     .inputChannel(incomingRequestsFromMaster())
                     .outputChannel(outgoingRepliesToMaster())
                     .build();
   }

   // Middleware beans setup omitted
}
```

This new annotation and builders take care of the heavy lifting of configuring infrastructure beans. You can now easily configure a master step as well as a Spring Integration flow on the worker side. You can find a remote chunking sample that uses these new APIs in the samples module as well as more details in the Spring Batch Integration chapter.

Just like the remote chunking configuration simplification, this version also introduces new APIs to

simplify a remote partitioning setup: `RemotePartitioningMasterStepBuilder` and `RemotePartitioningWorkerStepBuilder`. Those can be autowired in your configuration class if the `@EnableBatchIntegration` is present as shown in the following example:

```
@Configuration
@EnableBatchProcessing
@EnableBatchIntegration
public class RemotePartitioningAppConfig {

   @Autowired
   private RemotePartitioningMasterStepBuilderFactory masterStepBuilderFactory;

   @Autowired
   private RemotePartitioningWorkerStepBuilderFactory workerStepBuilderFactory;

   @Bean
   public Step masterStep() {
            return this.masterStepBuilderFactory
               .get("masterStep")
               .partitioner("workerStep", partitioner())
               .gridSize(10)
               .outputChannel(outgoingRequestsToWorkers())
               .inputChannel(incomingRepliesFromWorkers())
               .build();
   }

   @Bean
   public Step workerStep() {
            return this.workerStepBuilderFactory
               .get("workerStep")
               .inputChannel(incomingRequestsFromMaster())
               .outputChannel(outgoingRepliesToMaster())
               .chunk(100)
               .reader(itemReader())
               .processor(itemProcessor())
               .writer(itemWriter())
               .build();
   }

   // Middleware beans setup omitted
}
```

You can find more details about these new APIs in the Spring Batch Integration chapter.

1.3. JSON support

Spring Batch 4.1 adds support for the JSON format. This release introduces a new item reader that can read a JSON resource in the following format:

```
[
  {
    "isin": "123",
    "quantity": 1,
    "price": 1.2,
    "customer": "foo"
  },
  {
    "isin": "456",
    "quantity": 2,
    "price": 1.4,
    "customer": "bar"
  }
]
```

Similar to the `StaxEventItemReader` for XML, the new `JsonItemReader` uses streaming APIs to read JSON objects in chunks. Spring Batch supports two libraries:

- Jackson
- Gson

To add other libraries, you can implement the `JsonObjectReader` interface.

Writing JSON data is also supported through the `JsonFileItemWriter`. For more details about JSON support, see the ItemReaders and ItemWriters chapter.

1.4. Bean Validation API support

This release brings a new `ValidatingItemProcessor` implementation called `BeanValidatingItemProcessor` which allows you to validate items annotated with the Bean Validation API (JSR-303) annotations. For example, given the following type `Person`:

```
class Person {

    @NotEmpty
    private String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

you can validate items by declaring a `BeanValidatingItemProcessor` bean in your application context and register it as a processor in your chunk-oriented step:

```
@Bean
public BeanValidatingItemProcessor<Person> beanValidatingItemProcessor() throws
Exception {
    BeanValidatingItemProcessor<Person> beanValidatingItemProcessor = new
BeanValidatingItemProcessor<>();
    beanValidatingItemProcessor.setFilter(true);

    return beanValidatingItemProcessor;
}
```

1.5. JSR-305 support

This release adds support for JSR-305 annotations. It leverages Spring Framework's Null-safety annotations and adds them on all public APIs of Spring Batch.

These annotations will not only enforce null-safety when using Spring Batch APIs, but also can be used by IDEs to provide useful information related to nullability. For example, if a user wants to implement the `ItemReader` interface, any IDE supporting JSR-305 annotations will generate something like:

```
public class MyItemReader implements ItemReader<String> {

    @Nullable
    public String read() throws Exception {
        return null;
    }

}
```

The `@Nullable` annotation present on the `read` method makes it clear that the contract of this method says it may return `null`. This enforces what is said in its Javadoc, that the `read` method should return `null` when the data source is exhausted.

1.6. `FlatFileItemWriterBuilder` enhancements

Another small feature added in this release is a simplification of the configuration for the writing of a flat file. Specifically, these updates simplify the configuration of both a delimited and fixed width file. Below is an example of before and after the change.

```
// Before
@Bean
public FlatFileItemWriter<Item> itemWriter(Resource resource) {
    BeanWrapperFieldExtractor<Item> fieldExtractor =
            new BeanWrapperFieldExtractor<Item>();
    fieldExtractor.setNames(new String[] {"field1", "field2", "field3"});
    fieldExtractor.afterPropertiesSet();

    DelimitedLineAggregator aggregator = new DelimitedLineAggregator();
    aggregator.setFieldExtractor(fieldExtractor);
    aggregator.setDelimiter(";");

    return new FlatFileItemWriterBuilder<Item>()
            .name("itemWriter")
            .resource(resource)
            .lineAggregator(aggregator)
            .build();
}

// After
@Bean
public FlatFileItemWriter<Item> itemWriter(Resource resource) {
    return new FlatFileItemWriterBuilder<Item>()
            .name("itemWriter")
            .resource(resource)
            .delimited()
            .delimiter(";")
            .names(new String[] {"field1", "field2", "field3"})
            .build();
}
```